Ontology driven Enterprise Information Systems Engineering
Front-page:  
The picture on the front page is believed, but this is also challenged, to be originally an enigmatic German woodcut from the 16th century of further unknown origin. It may also be inspired by the book ‘Cosmographia’ by Sebastian Münster in the year 1544. The picture is about *Weltanschauung*, that what one may perceive as the World, one of several, maybe even unlimited, possible views of reality. There is also a reference to ”as above, so below”, which is one of the texts of the Hermetic Corpus, attributed to Hermes Trismegistos. Several elements seem to refer to the dawn of the ‘Age of Reason’. The version shown has been used also by Camille Flammarion, Paris, 1888.

* A missionary from the Middle Ages tells that he has found the place where Heaven and Earth meet...

The image first caught the attention of the author in a history book at high school. Now it is considered as the sudden discovery of a new *appropriate* and *truthful* representation of reality, two essential quality criteria for the study of ontologies. An ontology provides a new realm of insight and understanding. At the same time there is the understanding that reality can never be understood completely.

To my Nelly.
Ontology driven Enterprise Information Systems Engineering

Proefschrift
ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. Ir. K.C.A.M Luyben,
Voorzitter van het College voor Promoties,
in het openbaar te verdedigen op maandag, 3 december 2012, om 12.30 uur.
door
Steven Johan Herman VAN KERVEL,
elektrotechnisch ingenieur,
geboren te Arnhem.
Dit proefschrift is goedgekeurd door de promotor:

Prof. Dr. ir. J.L.G. Dietz.

Samenstelling promotiecommissie:

Rector Magnificus, Technische Universiteit Delft, voorzitter.
Prof. Dr. ir. J.L.G. Dietz, Technische Universiteit Delft, promotor.
Prof. Dr. E. Babkin, National Research University, HSE, Nizhny Novgorod, Russia
Prof. Dr. K. M. Van Hee, Technische Universiteit Eindhoven.
Prof. Dr. J. Iijima, Tokyo Institute of Technology, Japan.
Prof. Dr. J.B.F. Mulder, Universiteit Antwerpen, Antwerp Management School, België.
Prof. Dr. ir. A. Verbraeck, Technische Universiteit Delft, TBM.
Prof. Dr. J. Verelst, Universiteit Antwerpen, Antwerp Management School, België.
Prof. Dr. ir. H. J. Sips, Technische Universiteit Delft, EWI, reservelid.

SIKS Dissertation series nr. 2012-50
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Verspreid door: Formetis BV.
ISBN: 978-90-9027133-0

Druk: Drukkerij Mezclado, Tilburg
Omslag: Letterbeeld, Sander Neijnens, Tilburg.
Version V3-2k121217 - superceeds previous versions.
Acknowledgements

This thesis, as any thesis probably, is the result of a journey, an adventure and discovering 'things', 'ideas' and insights, rather than formulating first research questions and finding appropriate answers. The research questions have to be found and recognized and are often formulated afterwards, when the real value is understood. This finding is only possible if the circumstances are good, when one is surrounded by special people. This acknowledgement is devoted to some of these special persons. Each of these persons I admire very much for their specific qualities and I am very much indebted to them.

Special thanks go to my friends in science of the "CIAO!" Consortium. They inspired me and gave me guidance from their excellent knowledge and deeper insight. I admire each of them for their unique qualities. One of them told me, to my big surprise, that those generally appreciated powerful programming languages are actually a rich source of programming problems due to their excessive expressiveness. This led to the understanding of the value of minimized expressiveness of formal languages.

My special thanks go to my fellow adventurers in our professional life, Bob Zijlstra, John Hintzen, Joost Vermolen, Leon van Kervel, Mariejanne van der Heijden and Tycho van Meeuwen. We went through good and difficult times, as there are only difficult times in life and bad times do not exist. Without them, this research would not have been possible. In fulfilling the roles and responsibilities as paranymph in the defense of this thesis by Joost Vermolen and John Hintzen, their contribution is acknowledged rightfully. Especially I am grateful for the work of Sérgio Guerreiro, Tycho van Meeuwen and Bob Zijlstra, for their ruthless and merciless validation of the $\Psi$ processor simulations. Sérgio Guerreiro also showed in his Ph.D. thesis that the $\Psi$ processor provides the foundation of an enterprise control system, which supports this thesis.

Special thanks go to the members of the thesis committee. Each of them devoted me his expert scientific knowledge to assess rigorously this thesis and to improve it in many ways. I am grateful for their highly appreciated valuable support and help.

Special thanks are for my promoter Professor Jan L.G. Dietz, the founder of Enterprise Ontology and the DEMO methodology. He made a quantum jump to search and find the answers to the deeply hidden and difficult to grasp enterprise engineering problems in the roots of philosophy, ontology and the, for us engineers, seemingly esoteric empirical sciences such as sociology. The result is a radically different approach with on one side formally very strong and solid foundations. On the other side however, enterprise ontology is ultimately human oriented where human properties such as responsibility, authority, sincerity come first. This theory enables humans to work together in a human way. His insight changed my understanding of reality, the world of phenomena versus a personal
subjective interpretation. This emphasizes the need to verify and validate in a merciless and rigorous way one's worldview on the phenomena we experience in our daily life. And we must discard interpretations that are not appropriate or not truthful also in a merciless way.

The most important person in my life, is my Nelly, without her my life would not have its value, purpose and meaning, which is beyond any thanks.

Steven J.H. van Kervel, 2012.
Preface

In a way, this thesis is unusual. It is the result of an adventure in software engineering that started in the year 1993. It was in an application domain that could best be described by "the interactive production of tailor-made intangible services for a demanding customer". This domain was difficult to capture and several symptoms of now well known IT problems were already recognized. These problems involved the experience that IT projects fail too often and later do not support the organization well. The board of directors of a large banking and financial services company stated in 1997 that the design of a new financial service could be made within a few months but it would usually take several years to get the supporting IT systems ready.

At that time well-formulated pragmatic research questions did not exist; there was awareness of the questions "how to build better information systems" and "how to satisfy the customer", but not of any relation between the function and the construction of an artifact. Progress was made more by 'finding' small steps in understanding and improvement rather than by 'searching' in a systematic way.

When comparing this situation to the large scale manufacturing of tangible goods, such as cars, airplanes and electronic appliances etc, it was clear that in their application domain the IT engineers were doing much better work. Closer examination of the reasons for this apparent failure in our application domain revealed several differences we recognized at that time.

Logistic principles did not work or did not apply in our domain; there is no production in advance possible, supplies do not exist, there is no transportation since networks and later the internet would destroy distances.

The need for a tailor-made service\footnote{This applies especially for financial services and related complex legislation, where the customer must provide detailed information, take decisions, and must be informed completely and correctly.} requires the customer to act as an active co-producer of the product. This involves much communication and information exchange with an inexperienced co-worker.

The tailor-made intangible services have another degree of the internal complexity. While mass-produced tangible products may involve many, even millions of parts or components, the 'dimension' of complexity is low. All products are almost identical and manufactured in an almost identical way, aggregation of component parts. Complex customer services (especially financial services) are composed of fewer production steps, but each step may be conditional, and multiple paths exist for the outcome of the production process. A service may even be denied if certain conditions are not met. Support for rollback of production steps are mandatory. The execution path of production steps of services is highly unpredictable.
A new approach was chosen in 1993. It was recognized that a rigid executable program working with a relational type of database, the so-called client-server architecture, was not flexible enough to cope with the high degree of unpredictability during production. The idea of a “smart electronic dossier” was born. It would have both document qualities for the acquisition of produced information and be executable as a program. It would have “internal knowledge” about the execution of production steps and it would specify itself to be executed as an executable program. It had to be composed of software components that are aggregated and executed at runtime. It was implemented in our proprietary language DL/0, “document language zero”, which was in 1993 a proprietary representation of today’s XML. It was a model-instance driven software methodology where the model-instance is the executable program. The model-instance is its own type specification.

There was a shared and interactive modeling stage workshop with the knowledgeable stakeholders of the customer. During this stage, questions were being asked about the product, the customer and how the production should be made. Based on this information a working prototype software application was build within a few hours and together with the staff incrementally improved. Within typically five days around 70% of the functionality application would be implemented and accepted (excluding data conversion projects, interfaces to legacy IT systems). There were no programming specifications, the program was both the specification, the model and the implementation. We defended this approach by stating that this was “throw-away” software, not intended for software maintenance. This strength was also the main weakness, there were no reusable intermediate results.

At the beginning of this formal research project, a software suite and a set of matching business methods existed already for several years. The suite worked well in this professional domain but with several clearly recognized shortcomings; the three major problems of this research became more clear, notably the functional mismatch between the capabilities of the IT system and the needs of the organization. Though application development went very quickly, major rework was often needed.

It was recognized intuitively that a much better foundation exists when the theory of Enterprise Ontology and the DEMO methodology, developed by Dietz [2006] was found. It claimed to provide a significant better way to understand enterprises and delivered empirical evidence for appropriateness, “this works!”. In particular the claim for high quality formal specifications of enterprise models raised attention; "They say that it is formally correct, so this has to be good, though we don't understand why”. It was hypothesized that our software development platform would be suitable to build a software engine that executes the DEMO enterprise models in software. The decision to try this was then made and turned out to be successful. The software platform is now being rebuilt into a new generation and founded on the DEMO methodology. The recognition that the academic community, scientific methods and the formal rigor are essential quality criteria for this research led to the choice for a formal thesis.
The title of the thesis reflects this approach. The approach is the use of an ontology, in this case Enterprise Ontology, to drive the engineering of enterprise information systems. Engineering is the approach of construction of an artifact, “a working thing”, for a specific purpose. Enterprise information systems, the artifact to be constructed, are those information systems that support the operation of an enterprise.
Samenvatting

Ontologie gestuurde Constructie van Informatiesystemen voor Organisaties.

Achtergrond en overwegingen
De mens is een ingenieur, vanaf prehistorische tijden maakte hij op een creatieve manier gereedschappen voor een bepaald doel. Engineering is het proces van ontwerpen van de constructie en de werking van een artefact, afgeleid van de functionele eisen die aan het artefact gesteld worden en wel zodanig dat aan die eisen voldaan wordt. Deze twee verschillende perspectieven op ieder engineering artefact zijn de fundamenten van engineering en het onderwerp van dit onderzoek.

Gedurende de geschiedenis van de mensheid hebben mensen zeer indrukwekkende stenen bouwwerken voltooid; de Egyptische piramides de Chinese muur, de tempels in Midden-Amerika en de kathedralen in West-Europa zijn verbijsterende voorbeelden. Deze bouwwerken zijn tegenwoordig, zelfs met de nu beschikbare kennis en technologie, zeer indrukwekkend, verfijnd en moeilijk na te maken. De gebruikte kennis is zelfs deels verloren gegaan. Het zeer verstandige gebruik van materialen gevonden in de omgeving om de gereedschappen te maken om de stenen te zagen, op te tillen en te transporteren, waarschijnlijk alleen maar gebruikmakend van menselijke kracht en mogelijk die van dieren, is indrukwekkend. Dit zijn ingenieursprestaties die grenzen aan hetgeen mogelijk was.

In de Middeleeuwen wijdden de monniken in de kloosters hun tijd aan het kopiëren van de Bijbel en enkele andere zeer belangrijke boeken. Als kind op de lagere school waren we verbijsterd te horen hoeveel tijd, zorg, toewijding en geduld was besteed aan het overschrijven van de tekst en het maken van de prachtige illustraties. Er waren maar weinig mogelijkheden om fouten te verborgen of te corrigeren. Het betrof alleen maar kopiëren, de tekst en de achterliggende betekenis mochten niet veranderd worden. Dit kopiëren was de enige manier om godsdienst, cultuur en kennis te verspreiden en in die tijd speelden kloosters daarom een sleutelrol in de culturele en economische ontwikkeling van Europa. De hoeveelheid werk die nodig was voor de productie van een dergelijk boek was enorm; het was weer op de grens van wat mogelijk was.

In de tweede helft van de twintigste en het begin van de eenentwintigste eeuw is er nog een domein van taai en intensief werk dat met gelijksoortige absolute precisie moet worden uitgevoerd. Het is op het gebied van software engineering en het gaat om de kunst en de kunde om computer applicaties te maken die nuttige taken vervullen. De uitdrukking
“nuttige taak” verwijst naar het doel dat we willen bereiken met een dergelijke computer applicatie; het betreft een bepaalde gewenste functionaliteit. Eerst moeten we beschikken over de zogenaamde “functionele vereisten” of specificaties; die beschrijven wat de computer applicatie functioneel moet doen voor de gebruiker(s). Als we een computer applicatie hebben die voldoet aan de functionele specificaties dan is die applicatie in staat de “nuttige taak” uit te voeren voor de gebruikers. De functionele specificaties zijn subjectief, gedefinieerd en subjectief gezien door de ogen van de gebruiker(s) als zijnde geschikt om die nuttige taak uit te voeren. Zonder goede functionele vereisten kunnen programmeurs geen goed constructie ontwerp voor de samenstelling van de onderdelen en de onderlinge interacties daarvan, maken van die computer applicatie. Toepassing van de ingenieurspraktijk is het proces van het ontwerpen van de constructie en de werking van een “nuttig ding”, b.v. een computer applicatie, uitgaande van de functionele eisen. De constructie en de werking van een “nuttig ding” zijn objectief, in tegenstelling tot de subjectieve functionele vereisten. De objectieve constructie beschrijft alleen de samenstelling van de onderdelen en de interacties tussen die onderdelen.

We richten ons op organisaties en informatiesystemen die de operatie – werking van die organisatie ondersteunen. Organisaties zijn sociale systemen bestaande uit mensen die samenwerken aan een gemeenschappelijk doel of productie en communiceren over die productie.

De kwaliteit van computer applicaties, informatiesystemen, die organisaties in hun werk ondersteunen is vrij slecht. Gemiddeld faalt 65% van de projecten om die informatiesystemen te maken; de kosten overschrijden de budgetten, de functionaliteit van de informatiesystemen om de organisaties operationeel goed te ondersteunen is onder de verwachtingen en vaak wordt het hele project opgegeven voordat de applicatie gereed is. Programmeurs zijn wel in staat om zeer verfijnde, complexe en waardevolle computer applicaties te construeren zoals b.v. het internet, GPS navigatiesystemen, GSM telefoniesystemen, beeldverwerkende systemen voor medische diagnose, besturingsystemen voor efficiënte verbrandingsmotoren en nog veel meer. De inspanningen zijn vaak enorm maar ze slagen wel. De fundamentele reden dat dergelijke applicaties wel succesvol gemaakt kunnen worden is dat in die gebieden de onderliggende wetenschappelijke theorieën wel goed gefundeerd en geschikt zijn. Als die wetenschappelijke theorieën goed worden toegepast dan kunnen de hoogwaardige wiskundige modellen gemaakt worden die vereist zijn voor het maken van goede computer applicaties. Dit was nog niet het geval voor informatiesystemen voor de ondersteuning van organisaties. Solide wetenschappelijke kennis over organisaties ontbrak nog tot nu toe. We moeten eerst organisaties beter “begrijpen” door de toepassing van geschikte wetenschappelijke theorieën.

In dit onderzoek laten we de mislukkende projecten die veroorzaakt zijn vanwege politieke, strategische of financiële redenen buiten beschouwing. We richten ons op de drie belangrijkste problemen bij het maken van informatiesystemen voor organisaties zoals we
ze waarnemen, we identificeren de oorzaken daarvan en formuleren benaderingen om de
problemen aan te pakken. Om onduidelijkheid ten aanzien van dieper gelegen problemen te
voorkomen gebruiken we de term “probleem symptoom voor de problemen zoals we die
observeren.

I. Het eerste probleem symptoom dat we observeren, met veel ondersteunend empirisch
bewijs, is dat informatiesystemen voor organisaties de bedrijfsvoering, de werkwijze, niet
goed ondersteunen. Er is een functioneel falen van het informatiesysteem. We zien ook dat
pogingen falen (> 90%) om de bedrijfsvoering van een organisatie in overeenstemming te
brengen met een gekozen strategie voor een bepaalde markt, klanten, producten en
geldende wetgeving. Dit is ook een vorm van functioneel falen. De functie van de
organisatie voldoet niet goed voor het uitvoeren van de strategie. Deze beide functionele
problemen zijn nauw gerelateerd; een functionele mismatch van het IT systeem zal het
functioneren van de organisatie om de strategie uit te voeren verslechteren.

De hypothese die we formuleren is dat de belangrijkste oorzaak van deze problemen
gelegen is in het feit dat we niet goed in staat zijn functionele eisen op te stellen voor de
bedrijfsvoering van een organisatie en de ondersteunende informatiesystemen.

Een organisatie in werking is een zeer complex systeem met vele interne interacties en
afhankelijkheden, te complex voor onze ‘kleine hoofden’ met beperkte cognitieve
capaciteiten. Van business consultants wordt verwacht dat ze hoge kwaliteit functionele
eisen formuleren om de bedrijfsvoering van de organisatie zo goed mogelijk te
ondersteunen. Deze formuleringen blijken te vaak niet consistent, overcompleet, niet
alomvattend of niet coherent te zijn. De formuleringen bestaan vaak uit “illustraties” en
gaan dan vergezeld van verdere uitleg in vage en ongedefinieerde termen en dubbelzinnige
natuurlijke taal.

Uit brede en algemene ervaring kunnen we drie gedetailleerde constructie specificatie
problemen (dan wel symptomen van dieper gelegen problemen) voor programmeurs
identificeren:

i.) De specificaties specificeren niet één en slechts één correct geïmplementeerd
informatiesysteem. De vage, incomplete en dubbelzinnige specificaties leiden er toe
dat programmeurs zelf aannames moeten maken hoe de software geïmplementeerd
moet worden. Dit is buiten hun domein van competentie en leidt daarom tot
functionele mismatch van de software. De geboden functionaliteit van het software
systeem voldoet dan niet aan de functionele eisen.

ii.) Specificaties ondersteunen de programmeurs niet (goed) in de constructie van de
software. Programmeurs weten niet waar ze moeten beginnen en hoe de concepten
van de specificatie geïmplementeerd moeten worden in software in de gebruikte
programmeertalen.

iii.) Specificaties ondersteunen niet (goed) vroegtijdige validatie. Dit is de controle of het
software systeem zal functioneren in overeenstemming met de functionele

2 Opmerking van Edsger Dijkstra, (1930-2002), computer scientist en wiskundige. “We have small
heads, it is hard enough to understand one level of a function...” about understanding recursion.
Samenvatting

specificaties. In de regel ontdekken we de echte problemen van functionele mismatch pas als het software systeem na hoge kosten in acceptatie tests gaat, of zelfs later in productie. We moeten in staat zijn de constructie specificaties veel eerder te valideren, al gedurende het ontwerp van de constructie specificaties door middel van simulaties, gevolgd door cycli van herontwerp tot een goede functionele match is bereikt.

Deze constructie-specificatie problemen resulteren in de regel in een functionele mismatch die weer leidt tot herhaalde grootschalige revisies in de software, zoals beschreven bij probleem symptoom III.

We zien typisch in de praktijk dat alleen “kleine” informatiesystemen voor eenvoudige domeinen in de organisatie, niet te complex, zonder veel interfaces naar andere informatiesystemen, een redelijke kans hebben om correct gespecificeerd en geïmplementeerd te worden. Als gevolg daarvan zien we tegenwoordig “eiland IT systemen”, monolithische systemen voor kleine domeinen en zonder veel koppelingen naar andere systemen. In de regel faalt de integratie en interoperabiliteit van dergelijke systemen. De functionaliteit van dergelijke systemen vertoont vaak een ontbrekend gat of een overlap, hetgeen de reden is dat er zo veel redundante informatie is in dergelijke IT systemen. Redundante informatie in een organisatie is een bron van ernstige problemen en operationele fouten. Het correct ontwerpen van een IT landschap is een grote uitdaging.

II. Het tweede probleem symptoom betreft software programmering, met als gevolg onbeheersbare programmeerkosten. Dit is de vertaling van op hoog niveau geabstraheerde specificaties naar een gedetailleerde implementatie in een algemeen toegepaste programmeertaal. In het algemeen gebruikt een dergelijke taal data types als ‘real’, ‘integer’ en ‘string’. De op hoger niveau geabstraheerde concepten moeten uitgedrukt worden in deze primitievere data types en memberfuncties of aanroepbare methodes, hetgeen veel regels programmering vereist. Veel professionele software applicaties zijn extreem complex en kunnen uit 100.000 regels code bestaan, of zelfs veel meer.

De eerder genoemde problemen voor programmeurs (probleem symptoom I); i) ambiguïteit, incompleetheid en andere anomalieën in constructie specificaties; en ii) gebrek aan ondersteuning voor de feitelijke programmerfase; verergeren dit. De kosten voor programmering zijn in het algemeen enorm en oncontroleerbaar in termen van gebruik van middelen. Er is geen lineaire relatie tussen de omvang en complexiteit van het software systeem en de vereiste middelen. Als het software systeem groot en complex genoeg is zal het vrijwel zeker mislukken, tenzij we over hoogwaardige constructie specificaties beschikken.

III. Het derde grote probleem en uitdaging is dat IT systemen onderworpen zijn aan modificaties gedurende de levensduur, hetgeen naar ervaring steeds problematisch en duurder wordt. Er is een exponentiële groei in complexiteit van de software die onbeheersbaar wordt. IT systemen verzetten zich tegen aanpassingen en verslechteren. De term slijtage is van toepassing op de structuur van de programmatuur. Dit fenomeen wordt
Samenvatting

beschreven door en wordt aangepakt met de theorie van Normalized Systems. In dit onderzoek wordt dit probleem op een andere manier aangepakt.

Organisaties die opereren in een veranderende omgeving moeten “agile” zijn. Dit betekent dat een organisatie doorlopend zijn omgeving observeert, op veranderingen anticipeert, nieuwe strategieën ontwikkelt en toepast om zich op een flexibele manier aan te passen aan de veranderende omgeving. De uitdaging is dat IT systemen wel doorlopend mee gaan met de zich verder ontwikkelende organisatie zonder de noodzaak tot steeds complexere en duurdere software modificaties. Om aan deze uitdaging te voldoen moeten we IT systemen ontwikkelen op een dusdanige manier dat die niet verslechteren na doorlopende software modificaties.

Het probleem van de toenemende software complexiteit manifesteert zich al na de eerste implementatie als er een belangrijke functionele mismatch blijkt te bestaan. Cycli van uitgebreide modificaties kunnen nodig blijken te zijn voordat het IT systeem voldoet aan de functionele eisen. Iedere functionele modificatie vereist vele kleinere aanpassingen op verschillende plaatsen in de code hetgeen de complexiteit van de code sterk vergroot en de complexiteit voor een volgende wijziging vergroot. Na een aantal modificaties kan het systeem niet meer aangepast worden tegen acceptabele kosten en het project moet gestaakt worden voordat het in productie is gegaan.

Doelstellingen

De doelstellingen van deze research zijn om hoge kwaliteit IT systemen te construeren die geschikt zijn voor de agile organisatie en de oplossingen bieden voor de genoemde problemen. De problemen, de onderliggende probleembronnen en de geschikte benaderingen worden onderzocht:

i.) De bron van het probleem is “het niet goed begrijpen van organisaties”; we moeten eerst goed begrijpen wat organisaties zijn als fenomeen in de wereld. Dit manifesteert zichzelf door functionele eisen die al inadequaat zijn en de wortels van de latere functionele mismatch bevatten. Zelfs als we later hoge kwaliteit constructie specificaties afleveren en het IT systeem correct implementeren, dan zal het functionele mismatch probleem er gegarandeerd zijn. Om dit probleem aan te pakken moeten we wetenschappelijke theorieën toepassen om functionele eisen met een hoge graad van toepasbaarheid en getrouwheid op te kunnen stellen.

ii.) De problematische en oncontroleerbare programmering van complexe software, in het bijzonder als de constructie specificaties niet voldoen aan formele kwaliteitseisen; de specificaties zijn niet consistent, overcompleet, niet alomvattend of niet coherent. We moeten de bron van het probleem oplossen om systematisch hoge kwaliteit constructie specificaties op te stellen die aan die eisen voldoen.

iii.) Het probleem van de toenemende complexiteit en verdere verslechtering van software systemen na opvolgende modificaties. Zelfs als de constructie specificaties...
Samenvatting

voldoen aan de hoge kwaliteitseisen manifesteert dit probleem zich. De bron van het probleem is de inherente software complexiteit als de state of the art generieke programmeertalen toepassen. We moeten zoeken naar een methodologie waarbij software implementatie vanuit constructie specificaties essentieel eenvoudiger is. We moeten daarom de complexiteit van software programmering beteugelen.

Dit proefschrift levert een aanpak voor deze gerelateerde probleembonnen, gebaseerd op de volgende fundamenten:

i.) Het toepassen van geschikte en goed geaccepteerde wetenschappelijke theorieën en ingenieursmethodologiën. Dit omvat o.m. de $\Phi$, $\tau$, en $\Psi$ theorieën. In het bijzonder de toepassing van ontologiën is vereist om het probleem van “goed begrijpen” van de fenomenen die we waarnemen in de wereld der verschijnselen te benaderen. We passen ontologies ook toe als fundament voor hoge kwaliteit formele specificaties en het aanpakken van ontbrekende software specificatie kwaliteit. Voor organisaties passen we de theorie van enterprise ontologie toe. Voor andere domeinen moeten we andere ontologiën toepassen.

ii.) Het toepassen van formele methoden om hoge kwaliteit formele talen te specificeren om fenomenen in de wereld uit te drukken in modellen. Deze methoden omvatten ondermeer first order logic, de extended Backus Naur form voor contextvrije grammatica’s en formele taal theorieën. Het framework van Guizzardi voor ontologiën en structurele conceptuele modellen is toegepast om ontologische talen te construeren met bijzondere kwaliteiten en voordelen. Deze stellen ons in staat om de specificatie en programmeerproblemen aan te pakken op een radicale wijze.


Research Resultaten

Het eerste research resultaat is een engineering artefact, de GSDP-MDE methodologie, gebaseerd op het Guizzardi framework voor ontologiën en gestructureerde conceptuele modellen. Het omvat het ontwerp van een specifieke software engine en een taal voor ontologische modellen gebaseerd op een domein ontologie om bepaald deel van de wereld der verschijnselen te beschrijven. De GSDP-MDE methodologie is daarmee domeinonafhankelijk en universeel toepasbaar voor iedere domein ontologie. De ontologische modellen worden direct ingevoerd in de software engine en zijn dan ook direct executeerbare “broncode”. De software engine leest, schrijft en executeert de modellen uitgedrukt in de bijpassende ontologische taal. Na executie van het model waarbij het model zich modificeert, wordt het model weer gegenerereerd als broncode uitgedrukt in de taal en opgeslagen in een file repository. Dit impliceert dat het ontologische model altijd congruent is en blijft met de broncode van het software systeem. De software engine is een zgn. model-driven systeem en deze benadering wordt model-driven engineering (MDE).
Samenvatting

genoemd. Dit is mogelijk als zowel de systeem ontologie van de software engine als de ontologie van de taal isomorf zijn met de domein ontologie. Er is geen programmering meer (de afbeelding van ontologische concepten en relaties op de ‘primitieven’ en ‘constructs’ van de lagere generieke programmeertaal) voor dit domein. Het gerelateerde probleem van verslechtering van software gedurende opvolgende modificaties bestaat ook niet meer omdat er geen te onderhouden software meer bestaat. Herontwerp van een model is voldoende om een nieuw IT systeem te genereren. Er wordt een goede ondersteuning voor de agile organisatie geboden, een nieuw aangepast model levert direct een nieuw organisatie IT systeem op. De feitelijke inspanning is de implementatie van de software engine en de specifieke taal. Dit hoeft slechts eenmaal uitgevoerd te worden voor iedere domein ontologie en levert de hoogst mogelijke graad van software hergebruik op. Als de domein ontologie ook een hoge graad van geschiktheid en getrouwheid biedt dan wordt het probleem van de functionele mismatch goed aangepakt.

Het tweede research resultaat is de toepassing van de GSDP-MDE methodologie om enterprise informatie systemen (EISs) te construeren. De toegepaste domein ontologie is deel van de theorie van enterprise ontology, de Ψ theorie. De bijbehorende methodologie om enterprise modellen te construeren is DEMO7. Het resultaat is een set formele specificaties voor Ψ modellen. De specificaties omvatten de statische structuur en het dynamische gedrag.

Het derde research resultaat is de Ψ processor, de software implementatie van het tweede research resultaat. De Ψ processor is uitgebreid onderworpen aan kwaliteitsbeoordelingen, verificaties, simulaties en validatie. De Ψ processor is het fundament van de DEMO processor.

Het vierde research resultaat is het ontwerp, de eerste ontwerp cyclus, van de DEMO processor die alle vier DEMO aspect modellen volledig ondersteunt. Deze software engine

4 De primitieven van een formele taal zijn de symbolen zoals de woorden op papier in een natuurlijke taal. De constructs van een formele taal zijn methoden om relaties te creëren tussen primitieven, zoals b.v. vervoegingen voorgeschreven door de grammatica van een natuurlijke taal.
5 Opgemerkt wordt dat het heel eenvoudig lijkt, alleen maar een nieuw model en het nieuwe organisatie IT systeem is klaar. In de praktijk zijn er koppelingen met vele andere software systemen die ook aangepast moeten worden en waar deze voordelen niet behaald worden. Het geldt wel voor het ‘skelet’ van het IT systeem. Dit skelet is daarom direct geschikt voor simulatie en validatie in een zo vroeg mogelijk stadium, als het eerste model al beschikbaar is.
6 De meest gebruikte termen zijn “ontological appropriateness” en “ontological truthfulness”.
7 Afkorting van Design and Engineering Methodology for Organizations. 
8 Er worden bijzondere kwaliteitsvoordelen geclaimd door de onderliggende theorie. Deze voordelen worden empirisch gevalideerd in case studies.
9 De validaties omvatten de controle dat alle in de realiteit voorkomende fenomenen ook precies overeenkomstig voorkomen in de simulaties van de modellen.
10 Dit ontwerp is de eerste “design cycle” volgens het design science paradigm [Hevner, 2004].
Samenvatting

is geschikt voor DEMO model ontwikkeling, model simulatie, validatie en productie in een industriële toepassing.

Bijdragen aan Informatiesysteem Engineering

i) Eliminatie van programmering binnen het domein van de ontologie

Een DEMO model kan direct ingevoord worden in de DEMO processor zonder programmering. De DEMO processor is de software machine die models executeert, hetzij voor simulatie en validatie, dan wel in productie. Een tweede taal om DEMO modellen weer te geven is DMOL (DEMO Modeling Language) in XML representatie. Met de DEMO processor worden modellen ‘ge-edit’, geschreven en ingelezen vanuit achtergrond geheugen. Deze mogelijkheden pakken het probleem van software programmering en onbeheersbare kosten aan. Het probleem van verslechtering van de interne structuur van programmatuur wordt ook opgelost; er is geen programmering meer.

ii) Besturing en compliance van organisaties

Workflow is een prescriptief IT systeem dat conformiteit van een organisatie afdwingt volgens een model van de organisatie, in dit geval een DEMO model. Workflow capaciteiten worden hier volledig berekend uit de DEMO modellen en er is geen aparte BPM(-like) (Business Process Modeling) modellering nodig. The workflow capaciteit is ontologisch compleet, met “soundness” kwaliteit, afwezigheid van anomalieën en correct uitgevoerde roll-back fenomenen voor ongelimiteerd grote en complexe DEMO modellen.

iii) Functionele mismatch problemen

De mogelijkheid om DEMO modellen direct toe te passen voor vroege (zonder programmering) simulatie, validatie en incrementele verbeteringen door te voeren, pakt het probleem van functionele mismatch aan. Het is niet meer nodig om eerst het IT systeem te programmeren om de functionele mismatch problemen te ontdekken.

iv) Support voor die IT systemen die niet de organisatie ondersteunen


---

11 De algemeen gebruikte term is ‘truthfulness’.
12 De algemeen gebruikte term is ‘appropriateness’.
13 De ‘facts’ die tot stand zijn gekomen t.g.v. ‘acts’, beschreven in de Φ theorie (section 1.2.2).
14 SCADA is een afkorting van ‘Supervisory Control and Data Acquisition systems’. Dit zijn besturingssystemen voor productiestraten.
zgn. ‘adaptive model-driven case management systemen’ voor de productie van “intangible services” en “tangible” producten waarbij de klant een actieve rol speelt.

v) Ondersteuning van de ‘agile’ organisatie
Ondersteuning van de ‘agile’ organisatie wordt gerealiseerd door de eenvoud waarmee een nieuw DEMO model wordt ontworpen en direct toegepast kan worden als prescriptief informatie systeem.

vi) De eerste empirische ervaring
De eerste DEMO processor, zij het in een nog eenvoudige versie met beperkte functionaliteit, opereert als het centrale systeem van een adaptief case management systeem in een industriële productieomgeving (Mei 2012, Appendix IV). Dit resultaat toont de industriële toepasbaarheid aan. Het is de eerste casus waarin de genoemde symptomen van de problemen, i) de functionele mismatch, ii) de onbeheersbare software programmering en iii) de ondersteuning voor de agile organisatie, aangepakt lijken te worden. Veel meer empirische onderbouwing voor een dergelijke claim is vereist.

vii) Verwachte verbeteringen in IT systeem implementatie
Op basis van ervaring en inschatting zullen de vereiste middelen om IT systemen te implementeren gereduceerd worden met 50%. De ongewenste vrijheid van programmeurs wordt geëlimineerd. De productie van IT systemen uit modellen zal meer een commodity worden.

viii) Toekomstig onderzoek
Een aantal research onderwerpen worden voorgesteld die zich hoofdzakelijk richten op industriële toepassingen. Een aantal onderwerpen zijn voorgesteld voor interfaces, translaties en relaties met andere types informatie systemen, zoals MIS en productiebesturingssystemen.

Steven J.H. van Kervel.
Summary

Ontology driven Enterprise Information Systems Engineering

Rationale
Man is an engineer, from prehistoric times he made tools for specific purpose in a creative way. Engineering is the process of designing the construction and the operation of an engineering artifact, devised from the functional requirements, in such a way that the functional requirements are met. The two perspectives on any engineering artifact, construction and function are the foundation of engineering, and the subject of this thesis. Throughout the history of mankind, humans have constructed impressive stone buildings; the Egyptian pyramids, the Chinese wall, the temples in Middle-America, South-East Asia and the cathedrals in Europe are astonishing examples. These works are even by today’s standards, using the today's technology and mechanized tools, incredibly sophisticated and very hard to duplicate. Some of the knowledge how this has been done is lost. The very intelligent use of materials found in the neighborhood to make tools to cut, lift and transport the stones, using only the mechanical power of humans and probably some animals is impressive. These engineering achievements were at the edge of what was possible.

In the Middle Ages, monks in their monasteries devoted their days to copying the Bible and some other books. As children at the elementary school we were astonished to hear how much time, care, dedication and patience was spent on writing the text and making the beautiful decoration. There were few possibilities to hide or correct any mistakes. It was just copying; the text, the meaning was not to be affected in any way. This copying was the only way to disseminate religion, culture and knowledge and at that time, the monasteries played an essential role in the cultural and economic development of Europe. The amount of work involved to produce even a single book of almost perfect quality was huge; it was again at the edge of what was possible.

In the second half of the twentieth and the beginning of the twenty-first century there is another domain of tedious and elaborate work that must be carried out with similar absolute perfection. It is in software engineering, the art and science is to construct computer applications that perform useful tasks. The term “that perform useful tasks” describes our goal. First, we need to have so-called functional requirements. If we have a computer application that meets the functional requirements, then the computer application is able “to perform the useful task”. The functional requirements are subjective, “in the eye of the beholder”, defined by the perception of the user to solve some problem. Without good functional requirements software engineers cannot devise construction specifications, which is the composition of components and their interactions, of a software application.

We focus on enterprises and information systems that support the operation of an enterprise. Enterprises are cooperatives of human beings for delivering valuable results to
other human beings. More specifically, enterprises are social systems, i.e. the elements are social individuals (human beings) who cooperate and communicate about their productions.

The quality of computer applications for enterprises is rather poor. Approximately 65% of all enterprise software projects simply fail [Sauer et al, 2003]; the costs are exceeding budgets, the functionality is below expectations and often the whole project is abandoned before the applications become operational.

We, software engineers are however able to construct highly sophisticated, complex and valuable software systems. Examples are the Internet, GPS navigation systems, cell phone systems, imaging systems for medical diagnosis, control systems for efficient operation of car engines and many more. The efforts may be huge but we succeed. The fundamental reason that we are more successful in these areas is that the design and construction of these engineering artifacts is founded on appropriate, well-accepted and proven scientific foundations. Then we are able to devise the mathematical models we need for high quality constructional specifications for computer programs. This is not yet the case for enterprise information systems. Solid scientific knowledge about enterprises is lacking. We need first to 'understand' enterprises better by using appropriate scientific theories.

In this research, we do not consider project failures due to political, managerial, strategic or financial reasons. We assume that these non-engineering problem sources are not present. We focus on three major information system engineering problems as we perceive them, identify their sources and propose approaches to address the problems. These problems may have deeper hidden causes or problems. To prevent confusion with the deeper “problem sources” and “more deeper” problem sources, we call these our “problem symptoms”, as we experience these symptoms.

I. The first major problem symptom we observe, with much supporting evidence, is that information systems do not support the enterprise well. There is a functional mismatch problem. We also observe that efforts to align the operation of an enterprise with the developed strategy for a specific market, customers, products and compliance to regulations, usually fail (some reports: > 90%). This is also a functional mismatch problem; the function of the organization does not support the execution of the strategy well. These two functional mismatch problems are closely related; a functional mismatch of IT systems not well supporting the operation of an organization, will exacerbate a functional mismatch of the organization to execute the strategy well.

It is hypothesized that the most important cause of this problem is that we are often unable to devise high quality functional requirements for the operation of the enterprise and the supporting enterprise information systems.

An enterprise in operation is a very complex system with many internal interactions and dependencies, too complex for our ‘small heads’\(^\text{15}\) and limited cognitive capabilities. Senior

\(^{15}\text{Remark of Edsger Dijkstra, (1930-2002), computer scientist and mathematician. “We have small heads, it is hard enough to understand one level of a function...” about understanding recursion.}\)
business consultants are supposed to provide high quality functional requirement
documents for enterprise information systems that support the operation of the enterprise.
However, these specifications appear too often to be inconsistent, incomplete, incoherent
and incomprehensive. The specifications consist often of ‘illustrations’, accompanied by
further explanations in vague undefined terms and ambiguous natural language. From our
experience, we can identify three detailed deeper-rooted constructional specification
problem sources for software engineers:

i.) Specifications do not specify one and only one correctly implemented information
    system for this enterprise. Their vague, incomplete and ambiguous specifications lead
    the software engineers to guess how should be coded, which is outside their domain
    of competence and leads to functional mismatch, inconsistencies etc.

ii.) Specifications do not guide (well) software engineers in the construction of correct
     information systems. Software engineers do not know where to start or how to
     implement the concepts of these specifications in their programming languages.

iii.) Specifications do not support (well) early validation, which is the check if the
     information system will function the way it should and meet the functional
     requirements. We discover typically the real problems of functional mismatch after
     the systems have been implemented at high costs and go into in acceptance testing, or
     even later during operation. We should be able to validate specifications very early,
     already during the design of the specifications by simulation and re-engineer the
     specifications accordingly until a good functional match is achieved.

These constructional specification problems result typically in a functional mismatch,
which in turn leads to repeated major software overhaul efforts, as described at problem
symptom III.

We observe that typically only ‘small’ information systems for simple enterprise domains,
not too complex, without many interfaces to other information systems have a reasonable
chance to be specified properly and implemented well. Consequently we see nowadays
‘island information systems’; monolithic IT systems that cover small domains and that do
not interact and match well with other similar systems. Often the overall integration and
interoperation of all information systems fails. Functionality of these separate systems show
often a mismatch or an overlap, which is the reason why redundant information in different
information systems has become such a major source of problems in enterprises. Redundant
information is a major source of problems in organizations. The proper design of IT system
landscapes is a major challenge.

II. The second major problem symptom is in software programming and as consequence its
uncontrollable costs, even if we would have high quality specifications. This involves the
translation of high level abstract specifications of some information system into detailed
specifications, usually expressed in a general purpose programming language. Typically,
the programming language uses elementary primitives such as real, integer and strings. The
high level complex abstractions in the specifications must be expressed in these low-level
 primitives which may require many lines of complex coding. Many software applications are extremely complex and may constitute 100,000 lines of code or even much more. The before-mentioned problems for software engineers (problem symptom I); i) ambiguity, incompleteness and other anomalies in construction specifications and ii) lack of support for the actual programming phase; may exacerbate this problem symptom of programming and its uncontrollable costs.

The efforts of programming have typically become huge and uncontrollable in terms of use of resources. There is no linear relation between size and complexity of the software system and the resources typically needed. If the information system is complex enough, the implementation in software will almost certainly fail, unless we have high quality construction specifications.

III. The third major problem symptom and challenge to be addressed is that information systems are subjected to modifications and enhancements over time which becomes more and more problematic and expensive. There is an exponential increase of complexity in software over time that becomes unmanageable. Enterprises operating in a changing environment should be 'agile'. This implies that an enterprise continuously scans its environment, identifies changes, formulates new strategies and adapts itself to these changing conditions in a flexible way. The challenge of change and evolution exacerbates the before mentioned problems since information systems seem to resist ongoing modifications and adaptations. The challenge is that information systems continuously keep pace with the ever-evolving enterprise without the need for large, ever more complex and more expensive software modifications. To meet this challenge we need to be able to build software systems in such a way that they do not deteriorate over time after continuous modifications.

The problem of increasing software complexity manifests itself already after the first software implementation if there is a substantial functional mismatch. Several cycles of extensive modifications may be needed before the information system meets the functional requirements and can go into production. Each modification cycle involves many smaller modifications at many different but interdependent locations in the code and overall increase complexity of the code. After a number of software modifications, the software system cannot be modified anymore at acceptable costs for a given change and the project must be abandoned before going into production.

Objectives
The research objectives are to find reusable methods to facilitate the construction of high-quality enterprise information systems, fit for the agile enterprise, and provide solutions to the before-mentioned problem symptoms. The problems, the underlying problem sources and suitable approaches are investigated of:

i.) The problem source of not ‘understanding enterprises well enough’; we must understand enterprises well before we can specify functional requirements. This manifests itself by functional requirements that are already inadequate and carry the
The problematic and uncontrollable programming of complex software systems, especially if the construction specifications lack formal quality. Specifications are not consistent, not concise, not comprehensive or not coherent. We must solve the problem source of getting high quality construction specifications that meet these quality criteria.

iii.) The problem of growing complexity and further deteriorating of software systems after subsequent modifications over time. Even if the renewed construction specifications meet the high quality criteria after modification, this problem becomes manifest. The problem source is the inherent software complexity if we use the typical state of the art technology for programming. We must look for a methodology where software implementation from construction specifications is essentially simpler than typical technologies and programming languages. We must try to curb and defeat software programming complexity.

This thesis provides an approach for these closely related problem sources that is based on the following foundations:

i.) The use of appropriate and well-accepted scientific theories and engineering methodologies. These include the \( \Phi, \tau, \Psi \) theories. The use of ontologies is required to address the problem of ‘well understanding’ the phenomena that occur in the real world, specifically enterprises. We use ontologies also as the foundation to achieve high quality formal specifications and to address the problem of lacking software specification quality. For enterprises, we use the theory of enterprise ontology. For other domains we must use other ontologies.

ii.) The use of formal methods to design and specify high quality languages to express models of phenomena in the world. This foundation includes first order logic, the extended Backus-Naur form for context-free grammars and formal language theory. Also included are the framework of Guizzardi for ontology and structured conceptual models, and the law to be applied to design ontological languages with many benefits and high qualities. These qualities enable us to address the specification quality and software programming problems in a radical way.

iii.) The recognition that information systems are engineering artifacts and the systematic application of the engineering sciences, “how to construct artifacts that meet functional requirements”. The Generic Systems Development Process (GSDP), the Model Driven Engineering (MDE) methodology for software systems and Design Science Theory are applied.
Research Results

The first research result is an engineering artifact, the so-called GSDP-MDE methodology, based on the Guizzardi framework and the GSDP methodology. It captures the design of a dedicated software engine and a modeling language for ontological domain models. The ontological models are directly entered into the software engine as “source code”. The software engine reads, writes and executes the ontological models expressed in a matching dedicated high level language; i.e. the ontological model is congruent with the source code of the application. The software engine is a model-driven system and this approach is called Model Driven Engineering (MDE). This is possible if the systems ontology of the software engine and the ontology of the language metamodel are isomorphic to the domain ontology. There is no programming (which is the mapping of ontological models into primitives and constructs of a low-level generic programming language) anymore for this domain. The related problem of deterioration of software over time does not exist either because there is no programmed software to maintain and modify. Designing a new model is sufficient to generate a new information system. The support for the agile enterprise is enabled; a new model delivers directly a new enterprise information system. The remaining effort is the implementation of the dedicated software engine and the dedicated high level language. This has to be done only once for every domain ontology and offers the highest possible degree of software reuse. The underlying theory will not be modified. If the domain ontology exhibits also a good degree of ontological appropriateness and truthfulness, then the functional mismatch problem symptom of software systems is well addressed, the root cause is eliminated.

The second research result is the application of the GSDP-MDE methodology to construct enterprise information systems (EISs). The applied systems ontology for understanding enterprises is enterprise ontology and the accompanying methodology to construct models of enterprises is DEMO (Design and Engineering Methodology for Organizations, section 3.5). The foundation of enterprise ontology, the $\Psi$ theory, has been used to design the $\Psi$ processor. The result is a set of formal specifications of a software engine and an (XML) language to execute, construct (and destroy) $\Psi$ theory models.

The third research result is the $\Psi$ processor, the software implementation of the second research result for the $\Psi$ theory. The $\Psi$ Processor implementation has been extensively

---

16 It should be noted that it may seem to be very easy, just a new model and the new enterprise information system is ready. In practice there are many directly related software components that are linked to the enterprise information system and these may need to be overhauled too. It is true however for the skeleton of the enterprise information system. This skeleton enterprise information system is immediately suitable for simulation and validation exercises.

17 Once the dedicated software engine has been implemented there is no ongoing further development or maintenance necessary, since neither the ontology nor the GSDP-MDE methodology are assumed to change over time.
subjected to quality assessment\textsuperscript{18}, verification, simulations and validations\textsuperscript{19}. The $\Psi$ processor is the core of the DEMO processor.

The fourth research result is the design, the first design cycle\textsuperscript{20}, of the DEMO processor that supports the DEMO\textsuperscript{21} models completely. This software engine is suitable for model development, model simulation, validation, and production in an industrial environment.

**Contributions to Information Systems Engineering**

i) Elimination of programming within the domain of the ontology.
A DEMO model can be entered directly into the DEMO processor without any programming (translation to lower level primitives and constructs). The DEMO processor is the software engine that executes models, either for simulation and validation purposes, or for full production as an EIS. A second language to represent DEMO models is the DMOL (DEMO Modeling Language) expressed in XML\textsuperscript{22}. The DEMO processor edits models, writes and reads models expressed in DMOL to and from persistent memory.
These capabilities address the problem symptom of software programming and uncontrollable costs.
Also the problem of deterioration of IT systems over time is also addressed; since there is no more programming, there is no more deterioration of software. IT systems should be redesigned and implemented very quickly, supporting the agile enterprise.

ii) Enterprise compliance enforcement.
Workflow is a prescriptive information system that enforces compliance of the enterprise to the DEMO model of the organization. Workflow capabilities are completely calculated from the DEMO models and there is no need for BPM(-like) modeling. The workflow capability is guaranteed ontological complete, with soundness, absence of anomalies and correctly executed rollback phenomena for unlimited large and complex DEMO models.

iii) Addressing the functional mismatch problem
The possibility to apply DEMO models directly for early (without programming) simulation and validation addresses the problem symptom of functional mismatch of IT

\textsuperscript{18} The underlying Guizzardi - GSDP-MDE theories claim special language qualities. This is verified in case assessments.
\textsuperscript{19} The validation involves the check that all phenomena that occur in reality are also present in the simulations.
\textsuperscript{20} This design is the first design cycle according to the design science paradigm [Hevner, 2004]. This paradigm identifies repeated design cycles, each with an incremental improvement.
\textsuperscript{21} DEMO models are comprised of four so-called DEMO aspect models, each aspect model provides a different essential view on an enterprise.
\textsuperscript{22} Extensible Markup Language (XML) is a very flexible and convenient markup language. Developed by the World Wide Web Consortium (W3C) and widely standardized.
systems. This capability addresses also the problem of functional mismatch of the operation of the organization; the operation of the enterprise is compliant with the DEMO model and can be validated to check that the strategy is executed properly.

iv) Support for non-enterprise information systems
The DEMO processor provides truthful and appropriate information, all acts and calculated facts\(^\text{23}\) from the DEMO State model, to other IT systems. These are the so-called MIS (Management Information Systems) systems such as accounting systems, inventory control, CRM (Customer Relation Management) systems etc. The DEMO processor provides also an integration link to production systems, such as SCADA systems, document-based information systems etc. The DEMO processor is the foundation of so-called adaptive model-driven case management systems.

v) Support for the agile enterprise
Support for the agile enterprise is provided by the simplicity to design new DEMO models and apply these immediately as a prescriptive information system for the enterprise. The enterprise will comply immediately with the new enterprise model.

vi) Early empirical evidence
The first application of the DEMO processor, still a simplified version, is the core engine of a case management system that operates in a professional production environment (May 2012, Appendix IV). This result shows industrial feasibility of the developed four research results. It is the first empirical case where experience suggests that the before-mentioned symptoms of problems, i) functional mismatch; ii) uncontrollable software programming and iii) support for the agile enterprise; are addressed. Much more empirical evidence is needed to substantiate any claim.

vii) Expected improvements in information systems implementation.
It is expected, based on experience and estimations, that the resources required for implementation may be reduced on average by 50%. The unwanted programmers design freedom will be eliminated. The production of IT systems from enterprise models may become a commodity.

viii) Future research
Future research topics have been proposed that are mostly aiming at professional application in production environments. A number of future research topics have been formulated (section 8.10) to address the interfaces, translations and relations with other

\(^{23}\) The notions of ‘acts’ and ‘facts’ are described by the Φ theory (section 1.2.2).
kinds of information systems, such as MIS (management information systems) systems and production control systems.

Steven J.H. van Kervel.
CONTENTS

Part I Introduction

1. INTRODUCTION AND OUTLINE ................................................................. 3
   1.1. INTRODUCTION TO THE RESEARCH DOMAIN ......................................... 3
   1.2. SCIENCE PARADIGMS AND RESEARCH METHODOLOGIES ....................... 8
   1.3. THESIS OUTLINE .................................................................................. 13

2. RESEARCH OBJECTIVES, QUESTIONS AND APPROACH ......................... 17
   2.1. RESEARCH OBJECTIVES ...................................................................... 17
   2.2. RESEARCH QUESTIONS ....................................................................... 17
   2.3. RESEARCH APPROACH ....................................................................... 22

Part II Theoretical Foundations

3. THEORETICAL FOUNDATIONS OF INFORMATION SYSTEMS
   ENGINEERING .............................................................................................. 35
   3.1. INTRODUCTION ..................................................................................... 36
   3.2. INFORMATION SYSTEMS ENGINEERING AND ORGANIZATIONAL OPERATION .... 37
   3.3. INTRODUCTION TO ONTOLOGY .............................................................. 39
   3.4. ONTOLOGICAL LANGUAGES .................................................................. 45
   3.5. GENERIC SYSTEMS DEVELOPMENT PROCESS FOR MODEL DRIVEN ENGINEERING 57
   3.6. INTRODUCTION TO THE Ψ THEORY ...................................................... 65
   3.7. INTRODUCTION TO ENTERPRISE ONTOLOGY ....................................... 72
   3.8. THE DEMO METHODOLOGY AND DEMO MODELS .................................. 73
   3.9. RELATIONS BETWEEN THE DEMO ASPECT MODELS ............................... 77
   3.10. CONCLUSIONS AND RESULTS ............................................................... 79

Part III The GSDP-MDE Methodology applied to the Ψ Theory

4. DESIGN OF A STATIC MODEL FOR THE Ψ THEORY .................................. 85
   4.1. INTRODUCTION ..................................................................................... 85
4.2. \( \Psi \) Theory Derived Axioms and Propositions .................................................. 87
4.3. The Model Tuple and Axiomatic Specifications ................................................. 93
4.4. DMOL Representation of DEMO and \( \Psi \) Models ............................................. 113
4.5. Conclusions and Results .................................................................................. 123

5. Design of a Dynamic Logic Model for the \( \Psi \) Theory ......................... 125
5.1. Introduction ......................................................................................................... 125
5.2. The Standard Transaction Pattern .................................................................. 126
5.3. Aggregated Standard \( \Psi \) Transaction Patterns .............................................. 131
5.4. Transmission and Reflection Phenomena ......................................................... 136
5.5. State and Transition Space Specifications ......................................................... 139
5.6. Transaction Cancellation Patterns .................................................................... 170
5.7. Application of Business Rules to Transaction Patterns .............................. 182
5.8. Conclusions and Results .................................................................................. 182

6. \( \Psi \) Processor Validation, Verification and Quality Assessment ..................... 185
6.1. The \( \Psi \) Processor Role Based Control System for Enterprises .......... 185
6.2. Validation by Simulation .................................................................................... 191
6.3. DMOL Language Quality Assessment ............................................................... 194
6.4. Support for the C4-ness Claim of the \( \Psi \) Theory .................................... 198
6.5. Information Systems Capturing an Enterprise ............................................... 200
6.6. Conclusions and Results .................................................................................. 201

Part IV DEMO Processor Design and Construction

7. DEMO Processor Design and Construction .................................................... 205
7.1. Introduction to the DEMO Processor ................................................................. 205
7.2. DEMO Processor Capabilities and Design Requirements .......................... 211
7.3. Action Model Business Rule Calculations and Abstractions ..................... 216
7.4. Remaining Design Topics ................................................................................ 224
7.5. Conclusions and Results .................................................................................. 225

Part V Conclusions and Future Research

8. Conclusions and Future Research ..................................................................... 229
8.1. Research Questions Revisited ........................................................................ 229
Part VI References and Appendixes

REFERENCES ........................................................................................................... 253

APPENDIX I. Ψ PROCESSOR SIMULATION, VERIFICATION AND VALIDATION .............................................................. 269

APPENDIX II. Ψ PROCESSOR IMPLEMENTATION .................................................. 329

APPENDIX III. GLOSSARY, LEGENDS AND NOTATIONS .............................. 341

APPENDIX IV. THE DEMO PROCESSOR IN PRODUCTION ......................... 349

APPENDIX V. CURRICULUM VITAE ................................................................. 353
PART I

Introduction
1. Introduction and Outline

Therefore, just as water retains no constant shape, so in warfare there are no constant conditions.

He who can modify his tactics in relation to his opponent and thereby succeed in winning, may be called a heaven-born captain. \[6:32-33\]

The principles of warfare of General Sun Tzu’s 'Art of War', Written around 2500 years ago.

Enterprises compete like opposing armies in combat with their rivals; this is a condition for their prosperity. Sun Tzu, the famous Chinese general, advises us to nimbly change tactics under shifting conditions. Like an army, a modern enterprise must be capable of changing shape as easily as water does. Agility, flexibility, adaptability, but also quality, costs and service level, are essential developmental objectives and enterprise information systems should have the potential to enable their achievements. Hence, we can determine the practical and relevant aim of this research, directly derived from General Sun Tzu's advice:

Enterprise Information Systems must capture the operating enterprise in a truthful and appropriate way, and be able to evolve with it.

Abstract. The introduction and outline describe the research domain, the notions of enterprise, information systems, the research objectives, the applied science paradigms and methodologies, the research questions and the thesis outline.

1.1. Introduction to the Research Domain

The research domain of this thesis is the engineering of enterprise information systems. Enterprise information systems (EIS) are here precisely defined as information systems that support the operation of an enterprise, based on the notion of an enterprise provided in

---

24 Sun Tzu, also known as Sunzi and Sun Wu, is a Chinese general, author of a famous book on military thinking, strategy and business tactics. His insights are still regarded as highly relevant and valuable for many domains where winning, loosing and competition is essential.

25 There are many different definitions in the literature for EISs. For example, those applications that constitute an enterprise's existing system for handling company wide information; any applications that have an important enterprise-wide impact or applications that involve (only) business process model execution. These definitions do not apply here.
1. Introduction and Outline

this research (section 1.1.1) and notion of ‘operation’ as provided by the theory of enterprise ontology. An EIS supports the operation of the enterprise in a prescriptive way and provides also provides detailed information about the enterprise in operation. The relation between an EIS and the operation of an enterprise is therefor very close and bi-directional.

1.1.1. The Notion of Enterprise

Enterprises are cooperatives of human beings for delivering valuable results to other human beings. More specifically, enterprises are social systems26, i.e. the elements are social individuals (human beings). The operating principle of enterprises is that these individuals, commonly called actors, enter into and comply with commitments regarding the bringing about of products for the benefit of actors in the environment of the enterprise. They do so in communication and against a shared background of cultural norms and values. These commitments occur in patterns that follow the universal transaction pattern, which is a structure of coordination acts concerning one production act. A transaction involves two actors; one is the initiator and the other one the executor of the transaction. The organization of an enterprise is a network of actors and transactions [Dietz, 2006].

The relation between this notion of enterprise and for example a commercial company or an organization must be regarded with care. The commercial company is the “thing”, with all details and aspects, as we observe it in the real world. An enterprise is an abstract representation that includes exclusively those concepts that are specified by the theory of enterprise ontology, specified by Dietz [2006].

Enterprises are systems in the formal sense [Von Berthalanffy, 1969]; [Bunge, 1989]. There are two fundamental perspectives on systems (the τ theory, section 1.2.4) and thus on enterprises. The first is a functional perspective; the enterprises produce a results or production that satisfy the customers and the other stakeholders of the enterprise. The functional perspective is subjective, "in the eye of the beholder", in this case the customer and the stakeholders. Function is a relationship between a system and a stakeholder [Dietz, 2008].

The second perspective is the construction perspective: an enterprise is a social system, the elements are human actors. The actors produce (parts of) things that ultimately constitute the production of the enterprise. The term social refers to the fact that actors cooperate and communicate about their production. From the construction perspective, enterprises are designed artifacts that operate in some desired way to meet certain functional requirements, such as the delivery of valuable products. This thesis focuses on the construction and operation of enterprises and especially on supporting information systems.

---

26 This is the Habermas-based notion of enterprise. There are other notions of enterprise, which are not considered here.
1.1.2. The Notion of Enterprise Information Systems

Enterprise Information Systems\textsuperscript{27} (EIS) are defined here as those information systems that support, integrate and coordinate the operation of an enterprise, based on the notion of an enterprise as defined in the previous section. Enterprise information systems provide also a technology platform for the execution of business processes, and more. An enterprise service bus that connects and coordinates all data exchange for all other information systems is usually also included. This implies that there are other categories information systems in organizations that are not an EIS; for example financial accounting systems, customer relationship management systems, personnel systems etc. These information systems aggregate detailed data, typically from the EIS, and represent these to stakeholders, management etc. The EIS of an enterprise is however the foundation of these management information systems (MIS). Another category of non-EIS systems include production systems, control systems and data acquisition systems for industrial production plants, known as SCADA (Supervisory, Data and Control Systems). Throughout this thesis, we assume that the EISs we discuss are automated. Therefore, they are (also) referred to as IT systems.

An Enterprise Information Systems (EIS) serves at least two purposes. First, it provides each actor of the enterprise with appropriate and complete information to ensure that the actor operates according to the design of the enterprise. This involves the exchange of information within the enterprise. This implies that there is a design of an enterprise that is being used by the EIS to provide this information to all human actors. Second, the EIS provides information about the operation of the enterprise to all stakeholders, such as management and shareholders. This involves typically MIS type of information systems.

As stated before, an EIS supports the operation of the enterprise, which includes the execution of business processes and interconnection of all information systems.

1.1.3. Research Motivation and Objectives

This thesis is about the design and implementation of enterprise information systems. Current state of the art engineering methodologies that should result into high quality IT systems, fail too often. Too often one observes serious functional mismatches, pointing to a lack of business IT alignment, when the first release of an IT system is subjected to acceptance tests. The production of software systems is too often an uncontrollable and unmanageable process, there seems to be no reasonable relation between size, complexity and demand of resources in time and budgets. Moreover, software maintenance costs increase, after repeated major software modifications, at such a rate that further modifications become prohibitive expensive. These and other software engineering

\textsuperscript{27} We define this notion of an EIS, precisely based on the notion of an enterprise of section 1.1.1. because somewhat different definitions of EISs and enterprises exist in the literature.
problems collectively cause a gigantic waste of money and resources, all over the world. In this thesis, we focus on three major research objectives, that are our motivation.

The first research objective regards the first major problem symptom of the research objectives, the mismatch between the actual functionality of IT systems and the expectations of the stakeholders. This is a lack of business IT alignment since the enterprise is not well supported by the IT system. The IT Governance Institute [ITGI] defines alignment as the capacity to demonstrate a positive relationship between information technologies and the accepted financial measures of performance.

Apparently, business architects (assuming it is their task) are unable to produce high quality functional specifications of IT systems. If the information system would meet these functional specifications there would be no business IT alignment problem.

The second research objective involves the experience that IT projects do not deliver IT systems within controllable financial budgets and resources, and with a reasonable and constant ratio between size, complexity and costs [Sauer, Cuthbertson, 2003]. This is the consequence of uncontrollable software engineering with two underlying problem sources. The first is that from high level constructional specifications low level software instructions must be written, programming, which is already very complex and resource consuming, even with high quality construction specifications. The second problem source is construction specifications that lack quality, this will lead to unnecessary internal inconsistencies in the software and as a result a lacking business-IT alignment. This is usually discovered after expensive programming. The delivered software components need too often major overhaul cycles until some reasonable functionality has been achieved. These overhaul cycles requires again huge uncontrollable resources.

The third major research objective involves the observation that the costs of maintaining IT systems grow exponentially over time [Lehman, 1985]. The reason is that the quality of the internal structure of software deteriorates and the complexity grows; which makes subsequent software modifications ever more difficult. This problem is clarified by the theory of Normalized Systems [Mannaert, Verelst, 2008]. In this thesis a different approach to solve this problem is proposed.

These three major problems are widely recognized, with much supporting evidence, for example:

The Standish Group [2009] reports:

i.) 32% of projects succeeded on time, on budget and with required features and functions;
ii.) 44% of projects were challenged, late, over budget and/or were completed with less than the required features and functions;
iii.) 24% of projects were cancelled and never used. The Standish Group mentions also the size of a project as a risk factor; more than a budget of $10 million almost guarantees a failure. The size of the system and the size of the organization are also critical risk factors.

Budzier [Budzier, A, 2011] reports similar figures and refer to uncontrollable results and “black swan” type of unpredictability. Large IT projects have around 20 times more chance for failure than other business initiatives. They point also to political issues, which has been left out of scope. They report on average 55% more time and on average 27% more costs than anticipated. The management should anticipate a 400% budget overrun for large IT projects.

The Tata Consultancy [2007] (TCS) reports:

i.) 62% of organizations experienced IT projects that failed to meet their schedules;
ii.) 49% suffered from budget overruns;
iii.) 47% had higher than expected maintenance costs;
iv.) 41% failed to deliver the expected business value and ROI (Return on Investment);
v.) 33% failed to perform against expectations.

Saur and Cuthbertson [2003] find similar results.

The three problems of failing business – IT alignment, uncontrollable ICT projects and increasing software maintenance costs are interrelated and exacerbate each other. It is a practical experience that often the internal structure of IT systems has deteriorated already too much, beyond repair, even before the IT system can go into production acceptance tests. The project has to be abandoned then.

It is hypothesized (in section 3.2.2) that the cause of these problems is our failure to produce high quality IT system specifications. Political and managerial aspects are not within the domain of this research. The results of this research however may provide a more transparent view for political and managerial discussions. Current requirements engineering approaches fail in eliciting a complete and relevant set of functional requirements, covering all user expectations. In addition construction specifications lack quality which leads to the observed problems of uncontrollable software costs and software quality deterioration. This hypothesis is supported by the following observations:

When software engineers are provided with high quality constructional specifications, then they are usually able to construct high quality software systems, even if the application

---

28 The notion of “high quality” specifications, C4-ness quality, is addressed further in this section and in sections 2.2.1 and 6.4.
domain is complex. Examples are GPS systems, mobile cell phone systems and the very robust Internet. EIS specifications apparently lack sufficient quality. As a consequence, programmers have to take design and implementation decisions they should not take because they are not qualified to do this. There is unwanted design freedom and ambiguity. The business architects who produce the specifications should anticipate this. In general, IT system specifications lack comprehensiveness: everything that must be included is not there. Another shortcoming is the provision of irrelevant information, which is a lack of conciseness: everything that is not required should not be there. A third shortcoming is the lack of consistency: there are hidden anomalies or conflicting specifications. The fourth type of lacking quality concerns the observation that the parts of the IT system specifications do not match semantically. This is a lack of coherence. We conclude that the four quality criteria for IT system specifications we need are comprehensiveness, conciseness, consistency and coherence, abbreviated to C4-ness\(^{29}\). With C4-ness quality specifications software engineers are able to deliver high quality IT systems that perfectly comply with the constructional specifications.

However, this does not guarantee that a good business-IT alignment will be achieved; we may have high quality C4-ness specifications for a system that works flawlessly according to these specifications but does not meet fully the expectations. The activity to assure that a system complies in a functional way with the functional requirements is called validation. Since functional compliance assessment is in the subjective “eye of the beholder” of the stakeholders, their involvement is required.

To summarize, the research objectives are to address: i) the apparent functional mismatch or lack of business-IT alignment; ii) uncontrollable project costs, caused by the large programming efforts due to translation to low-level software instructions and a lack of quality of constructional specifications; and iii) the exponentially growing costs of software modifications over time which prohibits the support for the agile enterprise.

1.2. Science Paradigms and Research Methodologies

Information systems try to capture some part of reality in a meaningful and truthful way for all stakeholders involved. This implies that any theory and methodology for information systems must be based on the following science and engineering theories.

First, enterprises are phenomena operating in the real world and as such the empirical sciences, especially behavioral science and theories about human communication, must be understood and utilized. Stakeholders are investigating the design, construction and operation of enterprises and must be able to reason about enterprises. A meaningful, truthful, shared interpretation of the real world for all stakeholders is required. This is

\(^{29}\) C4-ness is the abbreviation of Concise and Coherent and Consistent and Comprehensive. In section 3.4.8 this notion is further detailed.
within the domain of philosophy, metaphysics and ontology. The use of ontologies is one of the foundations of this thesis.

Second, information systems make calculations by means of computers, using data collected from reality. This is within the domain of formal methods, logic and systems theory. Ontologies provide also a formal and explicit representation of the phenomena in the real world and provide the foundation of formal methods.

Third, in this thesis there are four engineering artifacts designed and constructed: a methodology, the design and specification of a software engine, the implementation of it in software and the design of a modeling suite for a specific application domain. These are based on the design science research paradigm, model driven engineering (MDE) and the Generic Systems Development Process (GSDP) (sections 1.2.8; 1.2.7; 1.2.6; 3.5).

The well-founded theories we apply are described below. Without these scientific foundations we are unable to solve the three problems mentioned in the Research Motivation, we are unable to meet the functional quality criteria of IT systems specifications and we are unable to construct the engineering artifacts.

### 1.2.1. Ontology

Ontology is part of metaphysics and encompasses the study of being, existence and reality. An ontology is defined as a formally correct set of abstractions or conceptualizations of reality, the relations between the conceptualizations and the attributes. Major contributors are Bunge [1989], Wand and Weber [1999]. An ontology provides models for so-called ‘shared reasoning between stakeholders’ of a given situation, a specific phenomenon in reality. Formal rigor, the quality that it is entirely founded in sound philosophical theories, avoiding incompleteness, inconsistencies and anomalies, is an invaluable asset. The fact that it is a formally correct representation implies (for our purpose) that it is not possible to construct different interpretations from a given ontological model\(^{30}\) (without making obvious errors in interpretations). Any potential ambiguity in interpretation of an ontological model is eliminated. Ontological models are expressed in some (formal) language. The set of symbols is the vocabulary of a language and the rules which specify the ordering of symbols is the grammar of that language. The grammar forbids the construction of models that cannot exist in reality. An important notion is ontological appropriateness, the extend to which the use of the ontology supports our practical purpose; it is hence a subjective notion. Section 3.3.1 covers the notion ontology further.

\(^{30}\) An important issue is that some ontologies capture complex phenomena with multiple aspects or views on the phenomena. A phenomenon must then be represented by a “single” model that is composed of these multiple views, or aspect models.
1. Introduction and Outline

1.2.2. The Guizzardi Framework for Ontologies and Conceptual Languages

Guizzardi [2005] investigated the relation between ontologies and structural conceptual models, expressed in conceptual (or ontological) languages. If there is a 1:1 cardinality relation between the concepts and their relations of the ontology versus the primitives and language constructs of the conceptual language, then valuable benefits are achieved for software engineering, discussed in detail in section 3.3.1. These advantages enable to address the fundamental problems of the research motivation (section 1.1.3).

1.2.3. The Φ Theory

The letter Φ is pronounced as ‘FI’, an acronym for Fact and Information about some ‘world’. The term world refers to a specific part of the universe we are interested in and of which we require factual information or knowledge. It is therefor also founded on ontology (section 1.2.1; 3.3) and discussed in detail [Dietz, 2006]. The part of interest of the universe may be for example ‘the world of bicycle repair’ or ‘the world of enterprises’. Any world of interest is assumed to be composed of Stata, Acta and Facta. Stata are those ‘things’ or phenomena in the world that seem to have existed from before the beginning of our observation. A fact refers to “something” that exists in reality and provides us with ‘factual’ knowledge about the world. Facts can be either concrete or abstract; a physical object is concrete while for example the ownership of an object is an abstract relation between two objects. Facta are those things or phenomena that are the result of Acta, actions or acts, undertaken by an entity. Facts are created (by acts) but they cannot be destroyed; they can be ignored but cannot disappear. Before any action or act is undertaken, the corresponding fact does not yet exist. Information is that what results in some meaningful thought, perception or concept in the human mind, as represented by the ontological parallelogram (section 3.3.3). The foundations of the Φ theory are also provided by the semiotic triangle (section 3.3.2) that relates signs and objects in the objective world and corresponding conceptualizations in the subjective world.

1.2.4. The τ Theory

The Greek letter τ is pronounced as “TAO”, an acronym for Technology, Architecture and Ontology [Dietz, 2006]. The τ theory relates to the construction, engineering and implementation of systems, artifacts that are designed to deliver some desired results. Enterprises are defined as social systems, comprising human individuals, cooperating to deliver some production to an outside human individual. From this definition it follows that there is a boundary between those individuals within the enterprise and those outside of it. There exists some interaction across the boundary of the enterprise. Systems theory, the study of a group of objects or subjects working in concert, first developed by von
1. Introduction and Outline

Bertalanffy\textsuperscript{31}, is a foundation of the theory of enterprise ontology. Based on the work of Bunge [1989], the $\tau$ theory defines what a system is. “Something” is a system if and only if there is:

\begin{enumerate}
  \item A composition $C$, a set of elements of some category;
  \item An environment $E$, a set of elements of the same category, that is disjoint with $C$;
  \item A structure $S$, a set of interactions between the elements of $C$ and between elements of $C$ and elements of $E$;
  \item Some production $P$ delivered by the elements of $C$ to the elements of $E$,
  \item An interaction $I$ between elements of $C$ and elements of $E$.
\end{enumerate}

Something is either a system or is not a system, it is an objective ‘thing’. It is fundamentally incorrect to subjectively “consider something as a system”. The interactions between the elements of $E$ are disregarded.

Other essential parts of the $\tau$ theory are the notions of white box model and black box model. The black box model represents the function perspective of a model, the use of the thing as it is “in the eye of the beholder”. The engineer cannot know in advance, at design time, with certainty how the artifact will be used. It might be totally different from his intentions, but still very useful for the beholder. The white box model represents the construction of a system. The $\tau$ theory is represented by the Generic Systems Development Process (GSDP) (section 1.2.6; 3.5).

1.2.5. The $\Psi$ Theory

The letter $\Psi$ is pronounced as PSI, the acronym of Performance in Social Interaction. The PSI theory [Dietz, 2006] includes four axioms and a theorem. The operation axiom, the composition axiom and the transaction axiom are shown in the research overview (figure 1.2) and investigated in detail in section 2.5. The fourth axiom, the distinction axiom and the organization theorem are not covered in this research. The operation axiom of the $\Psi$ theory states that in any organization, there are human actors, there is communication between these actors and the actors cooperate and interact to deliver some production. The transaction axiom states that there is communication between actors and this communication follows a precisely defined pattern, the transaction pattern. The composition axiom of the $\Psi$ theory states that productions are also themselves composed of other productions in a recursive way. The $\Psi$ theory is founded by the speech act theory [Austin, 1962], in social action theory [Habermas, 1981] and information systems theory. The $\Psi$ theory forms an essential part of enterprise ontology, discussed in section 2.6.

\textsuperscript{31} Ludwig von Bertalanffy, 1901 – 1972, biologist, author of the General System Theory (GST) or “systemics”. The GST is applied in many areas including engineering, cybernetics, biology, organizational theory, complex adaptive systems.
1. Introduction and Outline

1.2.6. The Generic System Development Process

The Generic System Development Process (GSDP) [Dietz, 2006] is based on the τ theory, describes the design process and provides a consistent terminology.

![Diagram of the Generic System Development Process](image)

**Figure 1.1.** The Generic System Development Process (GSDP) [Dietz, 2006].

This process is independent from any underlying theories and is applicable to all engineering sciences.

The GSDP relates functional design and constructional design, white box and black box models, for any type of system. In addition it shows the roles of architecture, ontology and technology in developing systems.

The GSDP is discussed in detail (section 2.2.1) for the Model Driven Engineering (MDE) approach of software systems in sections 1.2.7 and 3.5.

1.2.7. Model Driven Engineering

Model driven engineering (MDE), proposed by the OMG [OMG, The Object Management Group, 2001], is a software engineering approach. In MDE usually a model of a system is identical or congruent with the input, the 'source code' in a specific formal language. The “model == application” represents the essence of this approach in a simplified way. A

---

32 The term MDA, Model Driven Architecture, is a trademark of the OMG, is also used for model driven software. The notion ‘architecture’ is not clear and ambiguous, so the term MDE is being used.
software engine executes any model expressed in that language, either directly by interpretation or translation of the model to executable code through series of model transformations. An advantage of this approach is that a specific language can be designed for a specific purpose or application domain; the primitives and constructs of this language are “closer” to the concepts of the application domain than general purpose programming languages. This approach may offer specific benefits; simplifying the design and implementation of the model from requirement specifications and using the model for shared reasoning between stakeholders. The highest level is obtained if the primitives and constructs of the modeling language correspond directly to the concepts of the application domain.

MDE does not make any specific claims about the language qualities and implementation. A major cause of failures in applying MDE seems to be the lack of rigor in (formally) defining the distinct kinds of models, including the ignoring of the fundamental differences between functional and constructional models. These problems and the approach to address these are discussed in section 3.5.1. There are also important notion such PIM (Platform Independent Models), PSM (Platform Specific Models) and DSM (Domain Specific Models) investigated.

1.2.8. The Design Science Research Paradigm

Design Science Research [Hevner, 2007] is the embodiment of three closely related cycles of activities to design engineering artifacts and processes (also called meta-artifacts). The first cycle is the relevance cycle that delivers requirements from the contextual environment, the world of phenomena or the application domain. It bridges the environment with Design Science Research and delivers requirements, field testing or validation. The second cycle is the Design cycle where artifacts and processes are build, evaluated and optionally rebuild. The third cycle is the Rigor cycle where artifacts and processes are grounded in a knowledge base founded on scientific theories, formal methods and engineering methods.

1.3. Thesis Outline

Part I, Introduction
In Part I, sections 1 and 2, we identify the research domain of interest, the research motivation and the research questions. The before-mentioned research objective is to address the problems of EISs: i) the apparent functional mismatch or lack of business-IT alignment; ii) the uncontrollable costs of programming caused by a lack of quality of constructional specifications and problematic translation of specifications into executable code; and iii) the exponentially growing costs of software modifications over time which prohibits the support for the agile enterprise.
1. Introduction and Outline

We hypothesize (section 3.2.2) that the main cause of these problems is i) the lack of high quality functional requirements and constructional specifications for software systems, and ii) the problematic translation of model specifications into an executable program in a programming language. To address these problems we need a generic methodology for the design and implementation of software systems.

Part II, Theoretical Foundations

In Part II, section 3, we develop the domain-independent GSDP-MDE methodology for software systems. The GSDP-MDE methodology is based on a high quality ontology for a specific domain, the Using System ontology of the GSDP (figure 1.1). Formal methods and a set of laws are formulated and applied to construct i) a high quality ontological language to express ontological models and ii) a model instance driven software engine that executes models expressed in that language. The GSDP-MDE methodology is domain-independent, it is applied to enterprise ontology to meet the requirements of EISs and to address the problems. The GSDP-MDE methodology is the first engineering artifact of this thesis.

Part III, The GSDP-MDE Methodology applied to the $\Psi$ Theory

In Part III, sections 4, 5 and 6, we use the $\Psi$ theory as our domain ontology to understand enterprises. The axioms of the $\Psi$ theory are the foundation for the derivation of axiomatic specifications for the DMOL XML language that expresses $\Psi$ models, the DEMO Construction Model and the Process Model. Attention is devoted in section 6 to the quality assessment; to ensure that any DMOL software anomalies are eliminated such as construct excess, construct overload, and the benefits of minimized language expressiveness are kept. The formal specifications are the second engineering artifact. Section 2.4.1. provides an overview of application of the GSDP-MDE methodology applied to the $\Psi$ theory.

The third engineering artifact is the design and the implementation in software of the $\Psi$ processor, the software engine that executes DEMO models expressed in DMOL (Appendix II). The $\Psi$ processor allows execution of aggregated DMOL models; a DMOL model can be composed of a set of DMOL models. The composition can take place at runtime, controlled by the execution of DEMO action model rules. Similarly, model parts can be destroyed at runtime. The $\Psi$ processor is a runtime self-modifying software system. This capability represents the construction and destruction of transactions we observe in enterprises. The $\Psi$ processor, implemented in software, is subjected to extensive validation and verification simulations (Appendix I). There is a relevance cycle in validation. The validation includes specific phenomena that are observed in enterprises, such as nested transaction rollback for enterprises in production chains. The verification activities include

---

33 Political, financial, managerial and other non-engineering problem sources are out of our research domain.

34 DEMO stands for Design Engineering methodology of Organizations (section 3.8).
1. Introduction and Outline

anomalies, deadlocks, language quality evaluation, soundness criteria and model verification against the axioms of the Ψ theory. No anomalies have been found so far. Supporting reasoning for the C4-ness quality of enterprise ontology is provided. Also the capabilities of the Ψ processor as a role-based control system for enterprises, similar to a workflow system, are investigated.

Part IV, DEMO Processor Design and Construction
In Part IV, section 7, the Ψ processor is extended with the capabilities to support the DEMO methodology completely. All four DEMO aspect models, the Construction Model, the Process Model, the Action Model and the State Model, are completely represented in the (extended) DMOL language. The extended Ψ processor is the DEMO processor, a software development environment for the construction, simulation and execution of DEMO models. The DEMO processor offers complete workflow and BPM(-like) capabilities, eliminates anomalies and the need for BPM modeling. Workflow procedures are completely derived from DEMO models. The design of the DEMO processor is the fourth engineering artifact of this thesis.
In Appendix IV the first professional application of the DEMO processor is described.

Part V, Conclusions and Future Research
In Part V, section 8, the assessment of the research questions and suggestions of further research are formulated.
2. Research Objectives, Questions and Approach

Abstract. This section describes the research objectives. We must find solutions for some serious problems, or rather the symptoms of deeper problems, we experience when we design and implement enterprise information systems. These solutions are expressed by the three research questions. The research approach that has been applied follows the Design Science Paradigm that describes development cycles, guidelines and the types of artifacts that are delivered. Finally an outline of this thesis is provided.

2.1. Research Objectives

The main objective of this research is the design and construction of enterprise information systems (EISs) that support the operation of an enterprise. The research motivation identifies three major experienced symptoms of problems for any information systems, for any domain;

i.) The apparent functional mismatch or lack of business-IT alignment of software systems.

ii.) The problem symptom that engineering of software system is uncontrollable in terms of costs. The problem source is the complex process of software programming, the translation of high level specifications into executable code. The lack of quality of constructional specifications often observed, results in repeated overhaul cycles of the software system and increased costs.

iii.) The exponentially growing costs of software modifications over time prohibiting the support for the agile enterprise.

These experienced symptoms and underlying problems may exacerbate each other. For example, bad constructional specifications may cause several overhaul cycles with each time many modifications. These cycles further increase the costs for each overhaul cycle as described in item iii).

The objective of this thesis is to address these symptoms of problems and to provide a solution.

2.2. Research Questions

The first research question is how to acquire high quality IT system specifications and how to use the specifications to solve the perceived problems. The result of the first research question is the domain-independent GSDP-MDE approach.
If this research result has been achieved by the GSDP-MDE methodology, then we can apply the methodology to the design and construction of specific enterprise information systems. This constitutes the second research question.

The third research question addresses the detailed design and implementation of a software engine. This software engine should provide a solution to a certain extend to the problem sources of the research objectives.

2.2.1. The GSDP-MDE Approach Question

The first research question is how to obtain high quality IT system specifications to address the functional mismatch problem and how to use these specifications to construct IT systems and address the two above-mentioned, ii) and iii), problems of software programming.

The high quality of IT specifications has two perspectives. First the “internal” or C4-ness (section 3.4.8) qualities of the IT specifications that are required to eliminate ambiguity, inconsistencies etc and unwanted design freedom. Second the “external” qualities: specifications should be a truthful and an appropriate representation of the phenomena in the world. The approach is the use of a high quality ontology to acquire high quality IT specifications. It is assumed that an ontology exists for the domain of interest. A strong benefit of an ontology is that it supports shared reasoning of models between stakeholders, which is validation at a very early stage.

If we have specifications of C4-ness quality for the conceptual models, then it is possible to design a formal language that allows specifications to be represented as models expressed in the language. Then it is also possible to design a software engine that executes models expressed in the language. It is also possible to design this language and the software engine in such a way that conceptual models can be expressed directly in this language, without translation or programming into lower level software primitives and constructs. If software systems can be constructed automatically from conceptual models executed by a software engine, then we address the problem of high costs for software implementation in a very effective way; software programming is eliminated. We also eliminate the problem of software deterioration over time. Each time a modification is required, the information system is generated completely again from a modified conceptual model. There is no ‘history’ or legacy of older software versions. The only required effort is the design of a new, modified conceptual model.

---

35 The question can be raised if this software engine translates conceptual models to executable code in lower level generic programming languages. In this case the software executes the conceptual models directly, by interpretation, without any translation to lower level languages.
2. Research Objectives, Questions and Approach

Figure 2.1. Overview of the GSDP-MDE process for information systems engineering.

Figure 2.1, Adopted from Guizzardi [2005], and extended, shows that an ontology C should meet truthfulness and appropriateness quality criteria to avoid or minimize functional mismatches. The ontology should also deliver specifications that meet the C4-ness quality requirements. This is required to eliminate ambiguity, inconsistencies etc that lead to unwanted design freedom and uncontrollable software implementation. Next, specifications can be represented as models expressed in language L. The objective is to design a language L in such a way that specifications can be directly represented in L, which eliminates all programming efforts and programming related problems. Lastly a software engine executes models expressed in language L, which is a Model Driven Engineering (MDE) approach (section 3.5.1).

The engineering effort is now focussed on the design of the metamodel of language L and the software engine, which has to be done only once for a given ontology, e.g. enterprise ontology. One may expect in the future:\footnote{Paraphrase of Douglas McIlroy, Bell Laboratories, who stated in 1968 about software reuse: “Expect families of routines to be constructed on rational principles so that families fit together as building blocks. In short, [the user] should be able safely to regard components as black boxes.” Proposition pertaining to this thesis.}

Expect families of ontological languages and software engines
for specific domains in the real world,
where the language metamodel and the software engine ontologies
are isomorphic to the ontology that captures that domain.
These will be used to implement model instance driven information systems
for that domain.

The term ‘model instance driven’ means that the software engine executes model instances and that there may be many instances of a model simultaneously in execution in a production environment. The term ‘isomorphic ontologies’ is a result from the first research
question. The domain ontology has to be devised once, its appropriateness and truthfulness qualities established. The modeling language and the software engine have to be designed only once for this domain. The software engine has to be programmed once and only once; and does not need any modifications as long as the domain ontology does not change.

The first research question is formulated as follows:

*How to construct high quality IT system models, a high quality modeling language and an executing software engine, suited for any domain ontology?*

For this question we need to define a methodology, quality criteria and postulate laws to be applied. The specific benefits, including elimination of programming, must be shown. The result is the generic and domain-independent GSDP-MDE methodology. The GSDP-MDE methodology will be applied to the next research questions.

### 2.2.2. The Engineering of the Ψ Theory Finite State Automaton

The second research question is about how to construct enterprise information systems. From the GSDP-MDE approach follows that the choice of domain ontology for enterprises is the first step. We choose the theory of enterprise ontology to be the domain ontology for enterprises.

Until recently DEMO models are almost exclusively used for the understanding of enterprises and for creating a rigorous and correctly shared conceptualization for all stakeholders of the enterprise. Consequently stakeholders are able to better understand the enterprise, to reason about it and to (re)construct enterprise models after model validation. This is mandatory when enterprises have to be designed (or redesigned) [Mulder, 2006], as well as in the splitting or merging of enterprises [Op ‘t Land, 2008]. Enterprise ontology and enterprise engineering have also been combined with enterprise governance [Hoogervorst, 2009]. The application of DEMO has been shown to yield substantially better IT project specifications [Algoe, Boedhram, 2009] compared to conventional approaches. These empirical results support the ontological appropriateness and truthfulness claims.

The postulated axioms and the theorem of enterprise ontology, the C4-ness quality of enterprise models, the empirical evidence for the appropriateness and truthfulness claims and the availability of the DEMO methodology justify the choice for enterprise ontology as the domain ontology for enterprises.

However, the four graphical DEMO aspect models are not suitable to be read and written by a software engine. There is the need to develop another language, the computer-readable DMOL XML language, to express DEMO models. This is an engineering task based on the GSDP/MDE methodology.
The second research question is directed at: i) the design of a modeling language, DMOL, DEMO Modeling Language that is congruent with or isomorphic to the four graphical DEMO aspect models; and ii) the design of a software engine, which is a finite state automaton\(^{37}\), that executes enterprise ontology models.

The following requirements have to be met for our purpose:

i.) Any imaginable enterprise that may exist in the real world, is represented by one and only one DEMO model and one and only one DMOL model.

ii.) For any constructable enterprise model, represented in either DEMO models or in DMOL, there may exist an enterprise in the real world.

iii.) DMOL model construction involves the construction and destruction of model (parts) at runtime with full model inspection capabilities.

iv.) Any model under execution can be generated as a DMOL model instance, stored on persistent memory, and reconstructed with full state information.

Since an EIS captures an enterprise as a dynamic phenomenon, it follows that the EIS is a control system for that enterprise. The model under execution is always a truthful representation of the enterprise, including full state information.

v.) The absence of anomalies, ontological incompleteness, lack of soundness and deadlocks must be investigated.

vi.) It is necessary to validate that the DEMO processor captures always any phenomenon in an enterprise in a truthful way; there is a strict completeness requirement.

The second research question probes the possibility of translation of $\Psi$ models directly to software primitives and constructs, for execution by a software engine. The second research question is formulated as follows:

\[
\text{How to construct a finite state automaton, or a set of finite state automata, and a modeling language using the GSDP-MDE methodology, that is compliant to the $\Psi$ theory?}
\]

The $\Psi$ processor has been designed, implemented in software and subjected to extensive simulations. The above-mentioned requirements are assessed.

### 2.2.3. Design of the DEMO Processor

The third research subject is the design of tools and software systems to construct a DEMO processor for the execution of DEMO models. While DEMO is fully compliant with the $\Psi$ theory, there are many extensions to the $\Psi$ theory: a DEMO model is a much richer

\(^{37}\) It will be shown that if the model is composed of more than one transaction, the software engine is composed of a set of finite state automata.
specification of an enterprise than a $\Psi$ model. We need the DEMO processor to achieve the research objectives (section 2.1). Therefore the third research question is:

\begin{quote}
How to design the DEMO processor, based on the $\Psi$ processor, that supports all four DEMO aspect models?
\end{quote}

A DEMO processor is the enterprise ontology processor with adequate capabilities and functionality to construct, operate and specify DEMO models. There is the stringent requirement that all four DEMO aspect models must be supported completely by the DEMO processor. This question is primarily a software engineering question. It regards the construction of a convenient and practical development environment to work with DEMO models. A design of the $\Psi$ processor and the extensions for a DEMO processor are given in section 7.

2.3. Research Approach

We followed the design science research, described by Hevner et al. [2004, 2007], which appears to be the most appropriate research methodology in our case.

The design science paradigm seeks to support the design and construction of useful artifacts that can solve problems expressed as functional requirements. In order to achieve this it is necessary to gain knowledge and understanding of the whole problem domain where our specific problem resides as well as knowledge and understanding of the solution domain in which the artifact resides.

Design Science is about how to conduct, evaluate and present engineering artifacts. Hevner et al. argue that the empirical sciences apply also. The design of an enterprise as a social system is also an engineering artifact, so the behavioral sciences apply to in this research.

Hevner provides a concise conceptual framework and guidelines for engineering to describe and understand a problem domain and a solution domain. Additionally guidelines to construct an artifact and ways to assess its qualities as a solution for a problem are given. Problem solving capabilities include creativity and intuition, two characteristics that could be considered the most important for high quality solutions. These capabilities are non-deterministic, hence non-calculable, and remain the exclusive capability of humans.

2.3.1. Design Science Cycles

In Hevner [2007] a three-cycle view is presented where a design cycle, a rigor cycle and a relevance cycle are described. Design Science describes the two processes of building artifacts as a potential solution to a problem and then evaluating artifacts to check its value.
as a solution. These processes may be repeated a number of times in an iterative way to improve the solution.

i) The Design cycle
The Design cycle involves the design, the construction of an artifact; followed by an assessment of its function at an early stage. This assessment provides feedback for incremental modifications and enhancements of the artifact. Alternative design options can be investigated and rejected or accepted. This process is continued until an acceptable result is achieved.

ii) The Rigor Cycle
The Rigor cycle addresses the need for and identification of the use of proper scientific foundations and formal methods for the construction of the artifact. This rigor cycle allows to distinguish between science and (best) practice. The rigor cycle is implemented by the use of the scientific foundations (section 1.2).

iii) The Relevance Cycle
In Design Science Research the relevance cycle is an empirical assessment between the Environment of the artifact, involving requirements and field testing to validate if the objectives of this research are met. For this research several relevance cycles are identified.

2.3.2. Design Science Guidelines

i) Guideline 1, Design as an artifact
This means that any engineering artifact, see the four types, is a designed and purposeful solution to a problem. The Generic System Development Process (GSDP) [Dietz, 2006] describes the design process and provides a consistent terminology. This process is independent from any underlying theories and applies to all engineering sciences. GSDP relates architecture, functional design and constructional design, white box and black box models, for any type of system.

ii) Guideline 2, Problem relevance
This guidelines states that there has to be a real problem.

iii) Guideline 3, Design evaluation
Any engineering artifacts have to be evaluated to assess its qualities as a solution to a problem.
iv) Guideline 4, Research Contributions
Design Science should provide clear theoretical and methodological contributions to address the problems and/or provide specific solutions to a specific problem.

v) Guideline 5, Research Rigor
Rigorous formal methods are mandatory for the construction and evaluation of artifacts.

vi) Guideline 6, Design as Search Process
The design of an artifact is essentially a search process using available means. There may be non-deterministic processes; opposed to deterministic calculation of a solution.

vii) Guideline 7, Communication
The results of design science research must be presented in a suitable way for various audiences and stakeholders. In section 1.3 these guidelines are evaluated against the results of this research.

2.3.3. Design Science Artifacts
Design Science distinguishes four types of artifacts:

i.) Constructs; a vocabulary, symbols, grammars, etc.;
ii.) Models; abstractions and representations of the real world using constructs;
iii.) Methods; including algorithms and processes;
iv.) Instantiations; such as prototypes, implementations etc.

Five design science artifacts are identified for our research:

Artifact 1. The GSDP-MDE methodology (figure 1.1)
The artifact is a design science method with formal rigor, based on three postulated laws for the identification of high quality primitives and software constructs for ontological languages. The methodology addresses several software engineering problems; i) the minimization of programming efforts and related problems by minimizing programming entropy etc, ii) model construction with simultaneous model verification; iii) the optimization of software reuse by using high level software constructs; and iv) the optimal support for early model validation that addresses functional mismatch problems. The GSDP-MDE methodology provides the answer to the first research question.

Artifact 2. The Ψ processor and DMOL language specifications (figure 2.2. - vi)
The second research question addresses the design of the Ψ processor, which is a design cycle. This design requires specifications of primitives, constructs, state space and
transition space specifications. This is a design science construct composed of a vocabulary, symbols, a grammar, objects and their attributes and relations to represent \( \Psi \) models. The construct is represented in first order logic and algorithms suitable as programming specifications. The DMOL language is specified in EBNF (Extended Backus-Naur Form). Appropriate reasoning is applied to further support the existing claim for the C4-ness of enterprise ontology models. This artifact is the answer to the second research question.

Artifact 3. A \( \Psi \) processor software implementation (figure 2.2. - vii)
The \( \Psi \) processor with the DMOL language implementation in software is a design science instantiation of a working software prototype artifact derived from artifact 2. The main purposes are i) verification of the formal specifications (\( \Psi \) model & C4-ness verification, figure 1.2); ii) support for the truthfulness validation of enterprise ontology models: iii) ontological completeness validation (‘Does the \( \Psi \) processor support all possible phenomena observed in reality?’) and iv) the foundation of the DEMO processor. The \( \Psi \) processor is critical in the sense that its implementation should be flawless and perfectly correct. The verification of correctness in execution of the \( \Psi \) processor utilizes commonly known and pragmatic software verification techniques, but further extensive verification, also using temporal logic simulations, is needed. Validation reasoning has been applied to the phenomena that occur in simulated dynamic state changes of a \( \Psi \) model against observed dynamic state changes in enterprises. Validation reasoning has also been employed in phenomena that occur in simulated dynamic state changes of a \( \Psi \) model against compliance to the \( \Psi \) axioms. The simulations involve a Design Science Relevance Cycle for ontological completeness (item iii). This third artifact is needed to verify the formal correctness of the second artifact, the \( \Psi \) processor.

Artifact 4. The DEMO processor (figure 2.2. - x)
The DEMO processor is a design science instantiation, an extension of the \( \Psi \) processor. Many additional features support the construction and execution of DEMO models. The capabilities of the four DEMO aspect models are supported. The design covers the most relevant design issues in a pragmatic and professional way, resulting in a useful software development and production environment following implementation. This artifact is the answer to the third research question.

Artifact 5. The first DEMO processor application in production
In Appendix IV the first processor application in production is described. The first practical findings seem to confirm the anticipated results and address the research objectives (section 1.1.3). A good Business-IT alignment seems to be achieved. The application programming effort for the whole application, including interfaces and implementation of software parts outside the domain was “low”. Many more empirical cases are needed to substantiate or refute the claim that the objectives have been realized. The artifact type is a design science instantiation and the first results provide a design science relevance cycle.
In figure 2.2, an overview is given with precedential type of relations and part-of type relations for “domains”. Elucidation of the domains and subjects:

i.) Reality
Reality or the 'real world' is the world of phenomena, the things we observe around us, which we try to capture, understand and influence, and of which we are part of, including our ability to observe it. Reality “precedes” enterprise ontology; without reality there would not be any ontologies at all.

ii.) Enterprise ontology and four ontological aspect languages
For the study and understanding of phenomena observed in reality, ontologies are designed and used to understand and reason about reality (reality “precedes” ontology). An ontology is defined as a formal, explicit specification of a shared conceptualization of reality [Gruber, 1993]. The conceptualization is composed of concepts of objects, their attributes and their relations. The conceptualization is shared by knowledgeable human stakeholders, meaning in the first place that all stakeholders are assumed to have an identical understanding of each object or concept, each attribute and each relation. In the second place that each model, composed of these concepts, their attributes and the relations between objects or concepts has a shared identical interpretation by all stakeholders. In the third place an ontological model enables shared reasoning between stakeholders about the various qualities of that model and compare that to other models. Enterprise ontology is an ontology to model enterprises, developed by Dietz [2006] (section 3.4). The foundations of ontologies and static conceptual or ontological languages have been investigated by Guizzardi [2005] and others. These foundations have been extended to dynamic ontological languages in this research. Enterprise ontology models are dynamic and they require therefor a dynamic ontological language.

iii.) Reality
Reality or the 'real world' is the world of phenomena, the things we observe around us, which we try to capture, understand and influence, and of which we are part of, including our ability to observe it. Reality “precedes” enterprise ontology; without reality there would not be any ontologies at all.

38 The notion “domain” is here informal, and denotes some area of interest; from the context is clear what is meant.
Figure 2.2. Overview of the research domains and subjects for the application of the GSDP-MDE methodology applied to enterprise ontology (research artifacts 2, 3 and 4).
iv.) Enterprise ontology and four ontological aspect languages

For the study and understanding of phenomena observed in reality, ontologies are designed and used to understand and reason about reality (reality “precedes” ontology). An ontology is defined as a formal, explicit specification of a shared conceptualization of reality [Gruber, 1993]. The conceptualization is composed of concepts of objects, their attributes and their relations. The conceptualization is shared by knowledgeable human stakeholders, meaning in the first place that all stakeholders are assumed to have an identical understanding of each object or concept, each attribute and each relation. In the second place that each model, composed of these concepts, their attributes and the relations between objects or concepts has a shared identical interpretation by all stakeholders. In the third place an ontological model enables shared reasoning between stakeholders about the various qualities of that model and compare that to other models. Enterprise ontology is an ontology to model enterprises, developed by Dietz [2006] (section 3.4). The foundations of ontologies and static conceptual or ontological languages have been investigated by Guizzardi [2005] and others. These foundations have been extended to dynamic ontological languages in this research. Enterprise ontology models are dynamic and they require therefor a dynamic ontological language.

v.) The $\Psi$ theory

The Operation, Transaction, Composition and Distinction axioms constitute together the $\Psi$ theory, which is part of the enterprise ontology theory (section 2.5). Important note: The Distinction Axiom is irrelevant to the objective science paradigm on which software systems are based. There is no relation between this axiom and the $\Psi$ processor, $\Psi$ models etc. When referring in this research to the $\Psi$ theory only the Operation, the Transaction and the Composition axioms are meant and exclusively these axioms are the foundation of the $\Psi$ processor and $\Psi$ models.

vi.) Distinction Axiom and Organization Theorem

The Distinction Axiom and the Organization Theorem are also part of the enterprise ontology theory but exclusively apply to the process of ontological modeling.

vii.) DEMO methodology

The DEMO methodology [Dietz, 2006] is based on enterprise ontology; the four axioms and the Organization Theorem. DEMO is a precisely specified methodology to model enterprises, resulting in a set of four high quality DEMO Enterprise aspect models of an enterprise (section 3.7).
viii.) Ψ Model and DMOL language specifications
The Ψ model and DMOL formal specifications are derived directly from the Ψ theory using the GSDP-MDE approach (section 3). The static Ψ model formal specifications are investigated in detail in section 4. The dynamic specifications are investigated in section 5. The grammar of the DMOL (DEMO modeling language) language is specified in section 4.

ix.) Ψ processor engine
The Ψ processor is a finite state automaton, in fact it is a set of finite state automata. It is an implementation in software of the Ψ model formal specifications. The Ψ processor constructs, renders and executes Ψ models and reads/writes these models in the DMOL language to and from files in a repository. The Ψ Processor is the core engine of the DEMO processor.

x.) Ψ model & C4-ness verification
A Ψ model is a model of an enterprise that may exist in reality, compliant with the axioms of the Ψ theory. Because the Ψ theory is part of the enterprise ontology theory, any Ψ model is a limited (still lacking the enhancements and additional refinements of DEMO) version of a DEMO model. Similarly, the DEMO processor (section 7) is an enhanced version, where more functionality has been implemented, of the Ψ processor (sections 4 and 5) to capture all four DEMO aspect models. To verify the correctness of (i) the Ψ model specifications and (ii) correctness of the implementation in software (the Ψ processor), large scale simulations have been carried out. The Ψ processor is used for simulations, C4-ness claim verification and reasoning, soundness validation, and validation against the Ψ theory axioms (section 6.4, Appendix I).

xi.) Ψ theory truthfulness validation
The Ψ processor is also used to validate the ontological truthfulness of the Ψ theory, especially the dynamic phenomena that occur in large-scale simulations (section 7.2, Appendix I) and the aggregation of models into models to represent production chains.

xii.) DEMO processor
The DEMO processor is composed of the Ψ processor extended with capabilities to execute DEMO Action Model rules and calculation of the State Model facts (section 7). In the DEMO processor all four DEMO aspect models are completely supported, without loss or construct excess.
xi.ii.) Set of DEMO Enterprise Aspect models

DEMO Enterprise Aspect models, constructed using the DEMO methodology, are the point of departure for further detailed recursive and iterative DEMO modeling that represents one of the many implementations of the ontological design. This results in a set of extended models or DEMO software components that represent the detailed operation of the enterprise.

xiv.) DEMO software components

Directly derived from a DEMO model, a DEMO component is a software component extended with engineering and design decisions, to an executable model. A complex aggregated enterprise can be represented by aggregated sets of DEMO aspect models and derived DMOL models (fine-grained and recursive modeling, section 7.2.3).

xv.) Enterprise MIS, C, A & P systems

In a production environment there are interfaces to MIS, ‘C’, ‘A’ and ‘P’ type of IT systems. Enterprise MIS systems are IT systems that do not support directly the operation of an enterprise but aggregate and recalculate the atomic production facts calculated by the DEMO processor with external data and represent these in an appropriate way. The purpose is give an appropriate and truthful view of the ongoing operation of the enterprise. The ‘C’ system (communication world) calculates the communication world from the model and enforces compliance of the enterprise to the model under execution, which includes workflow capabilities. The ‘A’ systems (actors world) provide a mapping from actor roles in DEMO models to physical actors in an organization according to business rules and policies. The ‘P’ systems (production world) are those IT systems that capture the actual production in an organization.

39 Enterprise Ontology is based on the Φ theory (section 1.2.2) that states that there are Stata, Acta and Facta. The DEMO processor calculates the resulting facts from the DEMO State model (section 7.1.1) and reports these (optionally) to external MIS systems.
PART II

Theoretical Foundations
Lao Tse, Chinese philosopher, 6th century BC, assumed to be the author of the Tao Te Ching (or Dao De Jing, or Daodejing), hints to important insights. He points to the dead alleys in going further and trying to find more knowledge about the world. In this research we try to find knowledge of the world through the study of ontologies, conceptualizations, symbols and signs, subjective interpretation and intersubjective communication. This knowledge is valuable and allows to construct artifacts that meet functional requirements. However the knowledge we acquire in this way is abstracted, simplified and the result of a reductionist approach. His advice must be taken seriously and we should not wander away. We should be aware of the danger of a reductionistic approach of acquiring knowledge of the world through ontologies.

Abstract.
In this section, first the notions of enterprises, enterprise information systems, ontology, the \( \Phi \) theory are discussed in detail. Ontological languages based on the Guizzardi framework are investigated and offer valuable advantages that are not found in traditional general purpose programming languages. The first research question is how high quality information systems for any domain can be constructed that address the three major problem symptoms in information system engineering, as discussed in section 1.1.3; (i) functional mismatch or lack of business-IT alignment; (ii) uncontrollable and unmanageable costs of software programming and (iii) the exponentially increasing costs for subsequent software modifications and maintenance. A generic methodology founded on ontology, the Generic System Development Process (GSDP) and model driven engineering (MDE) is proposed. The first foundation is the use of domain ontologies and high quality ontological languages to express models. The use of a model being executed by a software system is the model driven engineering approach for software systems. The Generic Systems Development Process is applied. The result is the generic GSDP-MDE methodology to design information systems for any domain ontology. A set of three laws is
postulated that, if applied, guarantees the absence of anomalies, elimination of programming, elimination of complexity expressed as zero entropy, adequate and minimized expressiveness of the modeling language. The consequence is that a specific language must be constructed for this specific domain ontology and a software engine must be constructed to execute models. To apply the GSDP-MDE methodology for enterprise information systems the qualities of the domain ontology, in this case the theory of enterprise ontology, are investigated.

3.1. Introduction

In this thesis, the scope of interest is enterprises, cooperatives of human beings for delivering valuable results to other human beings (section 1.1.1). More specifically, enterprises are social systems whose elements are social individuals (human beings). The operating principle of enterprises is that these individuals, commonly called actors, enter into and comply with commitments regarding the bringing about of products for the benefits of actors in the environment of the enterprise. They do so in communication and against a shared background of cultural norms and values. These commitments occur in patterns that follow the universal transaction pattern, which is a structure of coordination acts concerning one production act. A transaction involves two actors; one is the initiator and the other one the executor of the transaction. The organization of an enterprise is a network of actors and transactions. [Dietz, 2006].

Next to the general term ‘enterprise’ we will use the term ‘organization’ when considering the construction of the enterprise and the term ‘business’ when studying the functions of the enterprise in operation.

The business orientation implies that people, utilities, activities and information systems function to solve some problem that is perceived or assumed by analysis from a business perspective. The phrase ‘purposefully designed social system’ refers to the fact that the design of an enterprise is an (engineered) artifact fulfilling a purpose, which is the actual production that should be delivered by the enterprise. The operation of an enterprise encompasses the activities of the actors and includes their interactions with the outside world.

40 The elimination of programming refers to the entering of models into the executing software engine without translation to lower level primitives and constructs within the domain.

41 The term zero entropy (section 3.4.7) implies that for any set of construction specifications there is one and only one implementation in software. The usually unwanted programmer’s “freedom” has disappeared.
3. Theoretical Foundations of Information Systems Engineering

3.2. Information Systems Engineering and Organizational Operation

In order for information systems to capture information about some part of the real world, a conceptualization of the domain of interest is made in computer readable and executable form. This conceptualization encompasses the state of the real world as well as any state changes. State changes are typically initiated by events in the real world.

Enterprise Information systems (EIS) are defined as information systems that support the operation of the organization of an enterprise. This implies that there may exist other information systems in organizations that are not an EIS, for example accounting systems, customer relationship management systems, etc. The EIS of an enterprise is however the foundation of these Management Information Systems (MIS). Throughout the thesis, we assume that the EISs we discuss are usually automated. Therefore, they are (also) referred to as IT systems.

3.2.1. The Purpose of Enterprise Information Systems

The mission statement of this investigation (formulated in section 1) is highly practical, inspired by Sun Tzu and directly derived from the observed fundamental problems of information systems engineering.

*Enterprise Information systems must capture the operating enterprise in a truthful and appropriate way and be able to evolve with it.*

The term truthful regards the correspondence between the real world and its model as captured by the information system. The implicit requirement of truthfulness is reflected in the confluence of two science paradigms as the foundation of this research. The term appropriate refers to the practical value of the provided information for the actors. The information system should not only produce truthful information but the value of the information should be appropriate for the operation of the enterprise. Whether an EIS captures an enterprise in operation in a truthful and appropriate way can be established only by empirical assessment of a sufficient number of business cases.

We use the notion of ontology to understand and represent static and dynamic phenomena in the real world. It is necessary to investigate exactly how to capture reality in a truthful and appropriate way.

The prosperous enterprise must be agile, continuously scrutinizing its environment and deciding how to evolve. This evolution is an unpredictable process, as it constitutes action based upon complex environmental influences; however an enterprise information system
should enable this fluidity and evolve appropriately. This research is also an approach to solve the challenges of enterprise agility and evolution of enterprise information systems.

3.2.2. Software Engineering Problems and Challenges

The state of the art engineering methodologies that should result into high quality IT systems, fail too often, as expressed earlier. Too often one observes serious functional mismatches, pointing to a lack of business IT alignment, when the first release of an IT system is subjected to acceptance tests. Moreover, software maintenance costs increase, after repeated major software modifications, at such a rate that further modifications become prohibitive expensive. These and other software engineering problems collectively cause a gigantic waste of money and resources, all over the world. We focus in detail on the three major problem symptoms that have been mentioned in the research motivation (section 1.1.3).

The first one regards the mismatch between the actual functionality of IT systems and the expectations of the stakeholders, which is a lack of business IT alignment [ITGI] (section 1.1.3).

The second problem symptom is that software programming in IT projects do not deliver IT systems within controllable financial budgets and resources (section 1.1.3).

The third major problem symptom is that the costs of maintaining IT systems grow exponentially over time.

From the point of software engineering, it is hypothesized that problem symptoms 1 and 2 come down to our failure to produce high quality IT system specifications. We support this hypothesis by the observation that, if software engineers are provided with high quality specifications, then they are usually able to construct high quality software systems. This is even the case if the application domain is very complex and the programming - translation from high level to low level concepts - is very complex. Examples are GPS systems, mobile cell phone systems, medical imaging systems and the very robust Internet. For each of these applications the specifications are complete and specified with formal rigor. EIS specifications apparently lack sufficient quality. As a consequence, programmers have to take design and implementation decisions they should not take because they are not qualified to do this. There is unwanted design freedom and ambiguity. The business architects who (are supposed to) produce the specifications should anticipate this. In general, IT system construction specifications lack comprehensiveness: which means that everything that should be included, must be there. Another shortcoming is the provision of irrelevant information, which is a lack of conciseness: everything that is not required should not be there. A third shortcoming is the lack of consistency: there are hidden anomalies or conflicting specifications. The fourth type of lacking quality concerns the observation that different parts of the IT system specifications do not match semantically. This is a lack of coherence. We conclude that the four quality criteria for IT system specifications we need are comprehensiveness, conciseness, consistency and coherence, abbreviated to C4-ness.
With C4-ness quality specifications software engineers are able to deliver high quality IT systems that perfectly comply with the specifications.

As the three mentioned problems of software programming, software maintenance and business-IT alignment apply to any IT system they must be solved in a generic, domain-independent way. This is expressed succinctly in the first research question:

*How to construct high quality IT system models, a high quality modeling language and an executing software engine, suited for any domain ontology?*

A methodology is proposed that is based on ontology, the GSDP [Dietz, 2006], the Guizzardi framework [Guizzardi, 2005], and model-driven engineering [OMG, The Object Management Group, 2001].

### 3.3. Introduction to Ontology

In this section, the notion of ontology, the phenomena in the real world (reality), conceptualizations, the representation of these phenomena by symbols, the interpretations of the representation by stakeholders and intersubjective communication between stakeholders, are discussed.

#### 3.3.1. Scientific Foundations of Ontology

Fundamental developments in semiotics and ontology have been achieved by Bunge [1989]. Bunge also includes reference to abstractions in mathematics, including deductive logic. This means that an ontology can be accompanied by a formal language; a language composed of a vocabulary and a grammar expressed in logic that is suitable to express models of phenomena in the real world. Also Wand et al. [1989] provided foundations of information systems from ontologies.

Practically, an ontology is a "formal, explicit specification of a conceptualization shared between stakeholders" [Gruber, 1993], though other definitions exist. It provides a shared vocabulary represented as a set (of limited size) of symbols of objects, which can be used to model some domain or part of the real world. It specifies (i) the type of objects that exist in the real world; (ii) the attributes or properties of the objects and (iii) the relations between the objects in a formal way [Arvidson, Flycht-Eriksson, 2008]. The term formal is an utmost important quality; there is one and only one correct interpretation possible of every model that is expressed in an ontology; any ambiguity is eliminated. If there is more than one interpretation possible then this ontology is flawed and not an ontology in the formal sense. There are other kinds of anomalies and quality aspects, to be discussed further.

---

42 Mario Augusto Bunge, born September 21, 1919, Buenos Aires, philosopher and physicist.
Ontologies are used in many fields such as artificial intelligence, the Semantic Web, software engineering, biomedical systems, library science, etc., as a form of knowledge representation about the world or some part of it. In the present research a dedicated ontology for engineering, operation and interoperability of enterprises, the theory of enterprise ontology [Dietz, 2006], is investigated. This constitutes one of the foundations of this research and is ubiquitous within it.

There are three important questions regarding the quality of an ontology. The first question is to what extent the concepts of the ontology are able to represent the reality in a truthful way for all stakeholders. This quality is the truthfulness, or ontological truthfulness, as postulated by Krogstie [2000]. Lack of truthfulness renders the ontology (almost) worthless.

The second question is about the ontological completeness, this is the question is every possible phenomenon in reality can be represented. Lack of ontological completeness also renders the ontology also (almost) worthless.

The third question, also from Krogstie about the quality of an ontology, is how well it supports understanding, reasoning and communication between stakeholders; this is referred to using the term comprehensibility appropriateness or ontological appropriateness. Here there is some purpose involved, the appropriateness depends on which purpose the ontology is being used. These three quality criteria, ontology in relation to the real world and ontology in relation to model interpretation (for understanding, communication and reasoning between stakeholders for some purpose), are essential for this research. If one of these qualities is lacking to any degree, then any results from further research have little value.

Strong support for the ontological truthfulness and appropriateness qualities of enterprise ontology has been provided by Mulder [2006] for rapid enterprise design; Hoogervorst [2009] for enterprise governance; Op’t Land [2008] for enterprise splitting and allying in addition to several others. In many cases these models have been primarily used for model interpretation and found empirically to be valuable and useful. Also for information systems that are directly derived from ontologies, the ontological truthfulness is essential; if not we would provide information systems that simply do not provide a truthful representation of the enterprise.

In the field of computer science there is another emphasis, the “other face” of ontology and also vital to our research. This emphasis is that an ontology is a formal representation of a set of concepts within a domain, the attributes of each concept and the relationships between those concepts. Attributes related to an object represent concepts such as weight, mass, volume, size, etc. It is used by stakeholders to reason about the properties of that domain, and may be used to define the domain. This is not a new approach [Smith, 2004], [Guizzardi, 2005], as S.H. Mealy had in the year 1967 already identified three different worlds in data processing:
3. Theoretical Foundations of Information Systems Engineering

i.) the real (or objective) world;

ii.) ideas about its existence in the minds of men;

iii.) symbols on paper or some other storage medium (shown in fig. 3.2).

Figure 3.1. Object in the objective world with abstractions and stakeholders conceptualizations in the subjective world (the symbols on a storage medium are not shown).

These three different worlds are consistent with the semiotic triangle (figure 3.2). If figure 3.1 is extended with signs or symbols then we reach the semiotic triangle. The focus in this research, expressed in the first research question (section 2.1), primarily regards this aspect of ontology: the engineering of information systems derived from an ontology.

The primary requirement of an ontology and ontology-based model is the need of a formal conceptualization of a part of the real world that is shared amongst human stakeholders (figure 3.1). Any description of the real world in some language is ambiguous, unless the language meets certain quality criteria. While it is possible to discover and identify differences in conceptualizations, it is not possible to verify with certainty that conceptualizations are identical.

3.3.2. The Semiotic Triangle

Ogden and Richards [1923] investigated the relationship between concepts, symbols and objects in the real world. This applies to many fields such as linguistics and semantics. In the Peirce 43 collected papers [1931-1958] is expressed that:

"A sign, or representamen, is something which stands to somebody for something in some respect or capacity. It addresses somebody, that is, creates in the mind of that person an equivalent sign, or perhaps a more developed sign. That sign which it creates I call the

43 Charles Sanders Peirce, philosopher, mathematician, logician, 1839-1914.
interpretant of the first sign. The sign stands for something, its object [or referent]. It stands for that object, not in all respects, but in reference to a sort of idea, which I have sometimes called the ground of the representamen."

Stakeholders use ontologies to share understanding of reality and reason about it. Stakeholders must create a conceptualization in their minds, which is done either by “pointing” to the physical objects or by interpretation of presented symbols (signs) and their ordering. The presented symbols are used instead of the physical objects. The semiotic triangle expresses clearly the relation between (i) the real world composed of objects; (ii) the abstract conceptualization in the heads of the stakeholders that reference objects and (iii) symbols that denote objects and designate concepts.

![Semiotic Triangle Diagram]

**Figure 3.2.** The semiotic triangle, indicating the relationship between the signs that denote (*) objects in the real world and conceptualizations of the stakeholder(s) in the subjective world. [Peirce].

There is a constructivist position about the phenomena in the world that states that the phenomena are the result of previous communication and agreements made between stakeholders. Stakeholders agree that a specific object is for example an apple and it is not a car. This distinction is the result of previous communication and agreement how an apple and a car look like and that a specific sign denotes that object. The semiotic triangle (figure 3.2) shows that each concept in the mind of a stakeholder or stakeholders, consistent with the intersubjective interpretation paradigm, is an abstraction of that object, and references that object. A sign designates a concept in the mind of the stakeholders and denotes the corresponding object in the real world. The symbols are represented using media such as paper, magnetic media as a carrier, together with ink dots, magnetic fields, etc., as modulations on that carrier, which are objects in the real world.

---

44 In fig. 3.2 in the Objective world any Sign denotes (an) Object(s). It should be noted that denotation is a subjective operation involving awareness, carried out in the Subjective world. It cannot be produced in the Objective world since Things and Objects have no awareness.
Figure 3.3 shows three worlds, the objective world, the subjective interpretation world for each individual stakeholder and the intersubjective communication world between stakeholders. The stakeholders communicate using signs (or symbols) in some language, in stead of pointing to the objects in the objective world. The concepts are considered to belong to the subjective world while the objects and their signs are considered to belong to the objective world.

At the current stages of technological advancement, there is no known way to verify objectively, to measure using instruments, that a certain conceptualization, derived either from the real world objects or from symbols, results in an identical interpretation in two different minds. Even the term identical is unclear and might be better replaced by the term ‘equivalent interpretation’.

This equivalent interpretation is formed by stakeholder communication or shared reasoning\(^\text{45}\) (figure 3.3) and this is represented in the linguistic theory of language and communication developed by Austin\(^\text{46}\) and Searle\(^\text{47}\), Habermas [1981], and further

\(^{45}\) In this research the intersubjective shared reasoning using models is a very important goal. For example, the DEMO models (section 3.6) are high quality models and the DEMO processor provides model simulation. This supports a better business-IT alignment.

\(^{46}\) Austin, J.L., 1911-1960, British philosopher.

\(^{47}\) Searle, J.R., 1932-, American philosopher.
3. Theoretical Foundations of Information Systems Engineering

described by Dietz and Widdershoven [1991]. Intersubjective communication of high quality is essential for the development of appropriate models.

3.3.3. The Ontological Parallelogram

The semiotic triangle that relates the notions of sign, object and concept, can be extended by the introduction of the notions type and class. The notions of ontology, type and class are discussed further in section 3.4. In this way, the notions of conceptualizations and objects (figure 3.1) and the subsequent introduction of class and types result in the ontological parallelogram (figure 3.4).

![Ontological Parallelogram Diagram](image)

**Figure 3.4** The ontological parallelogram.

A reference is a relation between a specific concept from the subjective world and a specific object in the real (or objective world), as shown in the semiotic triangle. A phenomenon in the real world is composed of one or more objects and as such these are referenced by one or more concepts from the subjective world. A concept is always a concept of a type and a type is hence a generic concept. An instantiation is a relation between a concept and its type, where the concept is the reference to a specific object and this object conforms to the type. Conformity is a relation between objects and a type; an object, or a set of 0, 1, ..., n objects, may conform to a type. All objects in the real world that conform to a certain type populate\(^{48}\) the class of objects. Class and type are closely related and correspond; a specific class is the extension of a specific type. The ontological parallelogram summarizes and represents the \(\Phi\) theory (section 1.2.2) [Dietz, 2006].

It is important to note that there are concrete objects, which are directly observable as tangible things. There are also abstract objects. A transaction between objects or the fact that a car belongs to a person are examples of abstract objects, while the car and the person are concrete objects. Abstract objects can only be communicated between stakeholders

\(^{48}\) The term “population”, instead of “populates”, is sometimes also shown, from “Class” to “Object”, with an equivalent meaning. The population of a Class is composed of a number of Objects that conform all to the same Type.
using signs, they are not observable by stakeholders as concrete objects. Both concrete and abstract objects are represented by facts and provide factual knowledge about our world. Example: the proposition “this object is a bicycle” is objective factual knowledge and “this bicycle belongs to John” is abstract factual knowledge.

Furthermore, we observe in the real world that some objects are similar in some way and share common properties and a type may hence be defined. An example is the class of cars; objects that belong to this class can be characterized by some properties which would include, for example, “there are components such as driver seats, engine, wheels, brakes and steering”. This is a “white box” or constructional specification (section 1.2.4) and any assessment whether an object conforms to this type specification or not, is straightforward, without any ambiguity. A type specification such as “metal object” is also a construction specification.

If the notion of cars would be used in a subjective way, for example “anything that is suitable for transportation” then it is a “black box” or functional specification (section 1.2.4). The conformity of an object to this kind of type is ambiguous since “it is in the eye of the beholder”. Functional type specifications may lead to serious and deeply hidden problems and are therefore to be avoided absolutely.

3.4. Ontological Languages

In the real world it can be observed that certain constraints exist between the objects we perceive. For example, an ontology about persons making up families, the situation where for two persons each is the parent of the other, does not exist. If a set of symbols or signs is being used to represent family members, it should be impossible to construct this impossible parent-child relation. There must be a set of constraints originating from the real world to prevent this.

Informally stated, a language can be used as a way to express something for interpretation and shared understanding between stakeholders. The “something” to be expressed can be anything in the objective world or subjective world. The expression about the “something” is made using a proposition composed of a sequence of signs (or symbols). Usually there is a linear sequence of signs, for example in natural languages or the usual programming languages. Optionally the structure can be two-dimensional or even more-dimensional. For example, the DEMO aspect models (section 3.6) are two-dimensional graphic representations of enterprises. A language must be “known” for any proper understanding of a proposition in that language. This implies that for all stakeholders the meaning of any symbol must be shared in an unambiguous way. The set of allowed signs of a language is called the vocabulary of the language. Any signs that are not member of the vocabulary have no meaning. The sequence of the symbols is also relevant since the interpretation of the proposition depends also on this sequence. A different sequence of the signs may lead to a different interpretation by stakeholders. The proper interpretation of a sequence of
signs of a proposition requires also knowledge about the allowed and non-allowed sequences of symbols. The allowed and not-allowed sequences are governed by a set of rules. The set of rules of a language is called the grammar of a language.

Natural languages for communication between humans are on one side very useful and suitable for all daily situations. These languages are also usually highly ambiguous and not suitable for precise communication, especially when we need very high quality expressions and interpretation by automated machines. For precise communication where any ambiguity in interpretation has to be eliminated, so-called formal languages are being used.

Expressions in formal languages, such as programming languages, can be very complex, demanding many signs to express something. A kind of higher quality languages are the so-called ontological languages where attempts have been made to minimize complexity in expressions as much as possible.

Guiizardi [2005] provides the definition of an ontological language:

If

i.) the vocabulary of a language $L$ is composed of symbols that represent atomic concepts of things of a domain in the real world; and

ii.) the grammar of $L$ forbids any specifications of phenomena that cannot exist in reality; and

iii.) any possible phenomenon or relation in reality is specified by one and only one specification in that language;

Then the language is called an ontological language. (figure 3.5).
3. Theoretical Foundations of Information Systems Engineering

Figure 3.5. Real world, conceptualizations and ontological modeling language with their corresponding instances [Guizzardi, 2005].

Guizzardi shows the relations between real world phenomena, conceptualization⁴⁹ (here ontology C) and language with their instances of a real world phenomenon m, its model M, and a specification S expressed in that language L (fig 3.5). It is an extension of the semiotic triangle (fig 3.2) where the references between reality and the symbolic modeling language are not shown.

The real world phenomena is the total set R of all possible instances and its atomic parts of phenomena m in the real world; it means that m populates R. At any point in time there is one and only one real world phenomenon m, excluding any others. Guizzardi uses the term “instance of” for the relation⁵⁰ between any m and R.

An abstraction of R leads to the conceptualization C, which allows to express any imaginable model M in such a way that any M corresponds with a real world phenomenon m; and vice versa, each m has an abstraction M and M is an instance of C.

The set C of conceptualizations is represented by the modeling language L; defined by the vocabulary and the grammar. Expressed in L, complying with the grammar and using symbols from the vocabulary, a set of specifications S can be constructed by the modeler.

The language L must have the qualities that it specifies all specifications S where for each S there is one and only one model⁵¹ M and one and only one real world phenomenon m, vice versa. This commutative cardinality law is postulated by Guizzardi [2005] as follows:

\[ m : M : S = 1 : 1 : 1 \]  

[Cardinality law 1]

The very strict \( m : M : S = 1:1:1 \) cardinality law for static ontologies is the point of departure for the investigation of dynamic ontologies. The suitability of a conceptual modeling language to represent a set of phenomena in the real world in a given domain (i.e. its domain and comprehensibility appropriateness) can be systematically evaluated. This is done by comparing the level of homomorphism⁵² between a concrete representation of the world view underlying the language (captured in a specification of the metamodel of the

---

⁴⁹ Guizzardi uses the term conceptualization C; we use also the term ontology C.

⁵⁰ Guizzardi uses the term “instance of” to relate a real world phenomenon m to the atomic things from R in the objective world, or model M and ontology C. This term is not to be confused with the term instantiation which is a relation between a concept and a type in the subjective world in the ontological parallelogram of fig. 3.4 in section 3.3.3.

⁵¹ The term ‘model’ is used sometimes in an ambiguous way. In figure 3.5 a model refers to a conceptualization M of phenomenon m. In general parlance and also in the following sections the specification S in language L is often also called a model. From the context is clear what is meant.

⁵² A homomorphism is a structure-preserving map between two algebraic structures. Several types of homomorphism exist.
language), with an explicit and formal representation of that domain, which is termed here a reference ontology.

3.4.1. Qualities of Ontological Languages with 1:1 Cardinality

Guizzardi [2005] presents the advantages of a modeling language which is just expressive enough (but nothing more than that) to capture any permissible state \( m \) in the real world, as well as being representative of any permissible state in the real world in any of its specifications \( S \). If the language \( L \) is too extensive, then specifications \( S \) can be constructed that cannot possibly occur in the real world. If the language \( L \) is too limited or narrow, then not all phenomena \( m \) can be represented in a specification \( S \). In most professional programming languages there are many or even an unlimited number of different ways of solving a computational problem in a functional and correctly executing way. In each of these different ways some abstraction of the real world is made, represented – programmed – by a specification \( S \) constructed in language \( L \) and is subjected to calculation by an automaton.

According to Guizzardi, the cardinality criterion, a specification \( S \) expressed in language \( L \), can be said to be ‘lucid’ with respect to a model \( M \) if the mapping from \( S \) to \( M \) is injective\(^{53}\). This means that each element or construct in \( S \) refers to not more than one entity or relation in \( M \). This is a measure for ontological clarity\(^{54}\). When the specification \( S \) is interpreted it is then immediately clear and without any ambiguity how the conceptualization should be constructed or interpreted. If elements or constructs in \( S \) refer to more than one entity, there is a higher level of complexity with the possibility to ambiguity and errors caused by misinterpretation. More information, in some form, would be needed to eliminate this ambiguity and increase overall complexity. The interpretation is that case not immediately clear.

The possibility of a single language construct \( L \) referring to more than one element in the model \( M \) that refers to more than one tuple entity, would also give rise to ambiguity. This is the possibility of constructing two or more different concepts from only one specification \( S \) which is defined as loss of ontological clarity of the language [Wand, 1989]. More information from some yet unknown source would be required to eliminate this ambiguity, resulting in an error-prone and unclear situation. Any loss of ontological clarity would disqualify the grammar as a basis of GSDP-MDE software engineering. For this reason it must be explicitly verified that any model entities and relations refer to one and only one primitive or construct of the language specification for a perfect ontological clarity.

\(^{53}\) Injective is a type of homomorphism, a structure-preserving map between two structures.

\(^{54}\) Ontological clarity refers to the quality that there is no ambiguity in the mapping.
There is also the reverse possibility of construct overload, in which a single specification construct $S$ represents two or more different model elements $M$. Construct overload is often used in programming languages where the programmer uses a single language construct and has the freedom to pass different sets and types of parameters. The compiler resolves the correct version of the overload by checking parameter sets and types. This often comfortable but dangerous convenience at the programmer’s level must be eliminated. This is achieved if there is a 1:1 cardinality between model elements and their relations on one side and language primitives and constructs on the other side (figure 3.6).

Figure 3.6. shows for an ontology of endurants, based on Guizzardi’s framework, the 1:1 cardinality between model elements and relations of conceptualization model $M$ and the language primitives and constructs of ontological language $L$.

This expresses the first cardinality law. If the requirements of figure 3.6 are met, then the ontology $C$ and the ontology of the metamodel of language $L$ are isomorphic and the language qualities are guaranteed. The process of constructing isomorphic ontologies is one of the parts of the application of the GSDP-MDE engineering approach.

---

55 An isomorphism is one of the homomorphisms, with a 1:1 commutative mapping for each element of one structure and the corresponding element of the other structure.
3.4.2. Ontology of Perdurants

The theory of Guizzardi [2005] is applied to ontologies of static entities, endurants, and to the 1:1:1 first cardinality law for static objects in reality, conceptualizations and representations in language $L$. There is no time dimension with states, or state transitions over time. However, in enterprise ontology (and all other dynamic ontologies) the phenomenon $m$ may exhibit a dynamic behavior. This must be represented by ontologies of perdurants, such that a specific enterprise may exhibit different states over time. Therefore, enterprise ontology is an ontology of perdurants.

This corresponds with the $\Phi$ theory (section 1.2.3). The world is assumed to be composed of Stata, Acta and Facta. Stata are those 'things' or phenomena that seem to have existed since the beginning of time or our observation. Facta are those things or phenomena that are the result of Acta, actions or acts, undertaken by an entity and part of the dynamic behavior of a system. Of interest are the change from the old to the new Facta, the state transition, and the resulting new Facta, the new state.

The following conditions are postulated for dynamic ontologies: (i) the allowed set of states for any phenomenon $M$ is discrete and limited in number; (ii) any state changes are deterministic; and (iii) any phenomenon $M$, composed of a limited number of entities and their relations, does not change over time. We assume a general and unspecified lack of anomalies. For each of the specific different states of the same phenomenon $m$ abstracted to the model $M$ and represented by the specification $S$ expressed in language $L$ (figure 3.5), the Guizzardi cardinality law should apply. In other words, imagine some real world phenomena, given by $m$ that changes its internal state over time without changing the elements and the relations. For these two states the first cardinality law applies. Therefore the very valuable language advantages (ontological clarity, minimized expressiveness and elimination of construct overload) of Guizzardi are kept for each of these two states.

The entities and their relations that compose the phenomenon $M$ are assumed not to change over time. Therefore any state and state change of phenomenon $M$ is represented exclusively by attribute value changes of the elements of model $S$. Due to the condition that the number of allowed states is limited, the number of attributes and the set of allowed attribute values is also limited. Any allowed state of model $S$ in language $L$ must be completely specified for all attributes and their values. There are state transition rules for each allowed state that either allow or prohibit the transition from the current state to any other state.

For ontologies of perdurants, the real world phenomena $m$ can be in a limited number of different states $i$; just as the models $M$ can be in a number of precisely corresponding states $i$ and the specifications $S$ of a model can be in a number of precisely corresponding states $i$. This is represented by the state space and transition space cardinality laws, postulated in addition to Guizzardi's first cardinality law.

---

56 This research is limited to discrete and deterministic systems and ontologies, with a limited number of concepts, states and transitions. We assume a general lack of anomalies in the ontology.
3. Theoretical Foundations of Information Systems Engineering

3.4.3. The State Space Cardinality Law for Ontologies of Perdurants

For dynamic phenomena \( m \), only the internal state of the phenomenon changes. The same applies to the dynamic models \( M \) and the specifications \( S \). The \( m : M : S = 1:1:1 \) cardinality is extended with:

\[
m[i] : M[i] : S[i] = 1:1:1; \text{ for any } i:1, \ldots, n; \ n \geq 1 \tag{Cardinality law 2}
\]

where \( i \) is an index for each corresponding allowed state for \( m, M, \) and \( S \). The integer number \( n \) of allowed states is limited. This law expresses that in reality, in the conceptualizations and representations in language \( L \), the allowed states correspond precisely. If \( n = 1 \) then cardinality law 2 'is simplified' to cardinality law 1 for an ontology of endurants. If \( n > 1 \), there is an ontology of perdurants. As postulated earlier, for each of the different states the first cardinality law applies and the Guizzardi language advantages are kept.

3.4.4. The Transition Space Cardinality Law for Ontologies of Perdurants

The state space cardinality does not specify the rules that allow or prohibit any state transitions for \( m, M, \) and \( S \). For example, the state transition from \( m[i] \) to \( m[j] \) with \( i \neq j \) is either possible or impossible in the real world. These state transitions must correspond also for \( m[i], M[i] \) and \( S[i] \). To express this, a Boolean function \( T(x, y) \) is defined that returns true for an allowed state transition from \( x \) to \( y \) and false for a prohibited state transition. Note that the function \( T(x, y) \) is an overloaded function that accepts parameters of different types. This is expressed as:

\[
T(m[i], m[j]) = T(M[i], M[j]) = T(S[i], S[j]) \text{ for any } i, j : 1, \ldots, n; \ i \neq j; \ n \geq 1 \tag{Cardinality law 3}
\]

The third cardinality law states that in reality, in the conceptualizations and representations in language \( L \), the allowed and prohibited state transitions also correspond precisely.

3.4.5. Software Complexity Reduction by High Level Abstractions

One fundamental source of problems of information systems is in programming, the complex translation of application specifications or models to implementation in source code and executable code. It is widely accepted that complexity reductions can be achieved through higher level abstractions in software languages; the expression of model elements in the primitives of the language becomes simpler. These high level abstractions improve also information systems maintenance, enhancements and evolution.
3. Theoretical Foundations of Information Systems Engineering

It is claimed that the highest level of information systems engineering is achieved when conceptual models of the real world are used to derive information systems automatically, minimizing or eliminating the gap between model and implementation in source code according to the GSDP-MDE methodology.

This claim is supported by the following reasoning:

i.) For any concept of the domain ontology there is one and only one software primitive.

ii.) For any relation between two concepts of the domain ontology there is one and only one software construct.

iii.) For any phenomenon in reality there is one and only one model expressed in the software primitives and constructs of the language.

Coding is reduced to 1:1 translation to software primitives and constructs. Any higher level of abstraction and derived improvements in complexity reduction of software do not exist.

3.4.6. Modeling Languages with Minimized Expressiveness

The current generation of general purpose programming languages and modeling environments are typically “too powerful”, “too large” and “too detailed”. Being “too powerful” means that there are more than one or even an unlimited number of ways to solve a specific problem. Being “too large” implies that ‘uninteresting and irrelevant’ programs outside of the present problem domain can be written too. Too “detailed” means that each program needs too many instructions for the implementation of an elementary part of the program specifications; an extreme example is coding in assembly language while a suitable high level language exists. A language with minimized expressiveness reduces complexity further which is a valuable advantage.

For example for DEMO models (section 3.6) of enterprises (section 1.1.1) minimized expression implies that:

i.) DEMO metamodels are not “too powerful”, there is one and only one DEMO model to represent an enterprise operating in the real world.

ii.) DEMO metamodels are not “too large”, verified DEMO models cannot express ‘anything’ that is not an enterprise; or anything that is ‘outside’ enterprise ontology.

iii.) DEMO models are not “too detailed”; there is one and only one symbol of the DEMO metamodel that corresponds with a specific element (actor, transaction etc) of an enterprise. If there would more than one symbol needed, the DEMO model would be “too detailed”.

In addition there is a completeness quality criteria; any enterprise that may exist in the real world can be expressed by one and only one DEMO model. The minimized expressiveness quality and the completeness quality are achieved if the ontology and the metamodel of the language \( L \) are isomorphic ontologies.

---

57 This applies to the Habermas-based notion of enterprise of section 1.1.1. There are other notions of enterprises in the literature, which do not apply here.
3.4.7. Complexity and Entropy of Models expressed in a Language

Complexity, chaos and entropy are closely related notions. Chaos refers to our apparent inability to specify and calculate states and processes. The notion of entropy is the foundation of the second law of thermodynamics, which expresses that a specific macroscopic state of a system may correspond to many different states at the microscopic level. The first investigations in the determining of this law were of gasses in a closed encapsulation at manipulated pressures and temperatures. At the macroscopic level the temperature and pressure of the gas, measured by our instruments, remained constant as long as no energy was added or withdrawn from the closed system. Conversely, at the microscopic level the Boltzmann statistics\(^{58}\) describe moving gas atoms exchanging energy, due to their collisions with one another, in a complex and chaotic way. The entropy describes the number of different microscopic states that correspond to a specific macroscopic state.

The notion of entropy is useful to describe the chaos and complexity in software engineering: for example the mapping of an enterprise model to a sequence of instructions using the instruction set of a central processing unit. If it is considered that a specific ontological model of an enterprise at a certain state of transaction execution forms the macroscopic state, accordingly the number of different specifications expressed in a language that describe this enterprise at the macroscopic level completely and correctly is the entropy of that model specification. It is clear that using for example assembly language provides an unlimited number of different, albeit correctly executing programs to describe the enterprise; even adding pieces of ‘dead code’ is enough to show this. The same applies for higher level languages such as Pascal and ‘C’. This unwanted design freedom allows for the exponential explosion of complexity due to subsequent software modifications as explained by the Normalized Systems theory [Mannaert, Verelst, 2008].

Formulating the relation between a number of micro-states that all correspond to a specific macro-state requires a certain amount of information. This information answers the question, “Which micro-states correspond to this specific macro-state?” The notion of information used here is that according to the Shannon information theorem [Shannon, 1948] in which information does not refer to any semantic meaning or interpretation by stakeholders, as is the case in other parts of the research on communication. This different notion of information refers to the minimal number of bits, or bandwidth, required to be transmitted in order to reconstruct a given message after transmission. It is a measure of the entropy in a message. In other words, the message contains a binary number that tells us precisely which micro-state we are dealing with.

\(^{58}\) Postulated by Ludwig Eduard Boltzmann, Austrian physicist and major contributor to the development of statistical thermodynamics, 1844-1906.
The Guizzardi 1:1:1 cardinality between an enterprise model and specification $S$, expressed in language $L$, implies that for any model in the real world at any state there is one and only one $S$. This implies that language $L$ is a zero entropy language with, \[ \text{entropy} = \log_2(N/n) = 0 \] with $n == N$; where $N$ is the number of macro-states and $n$ is the number of micro-states or language specifications $S$ (usually the symbol $S$ is used for entropy, however this has been avoided here to prevent confusion with a specification $S$). There is no exponential explosion of complexity when modifying a specification $S$ as a program. It is possible to eliminate syntactical programming errors using a well-designed model development environment. Program validation is as simple as possible, using a limited number of propositions, which is precisely enough to prove perfect program correctness. It is an extremely safe software development environment. This is a form of reuse such as is advocated in object orientation; the concepts are implemented in code only once for unlimited reuse. The elimination of constructing any models that cannot exist in reality offers a significant higher quality of reuse than those of object orientation.

The drawback to this formulation and approach is the development efforts for zero entropy ontological languages. A systems ontology must be developed, the appropriateness quality must be assessed, the truthfulness quality must be validated. The language must be constructed and an executing software engine must be constructed and implemented using isomorphic ontologies. The application domain of the language is as narrow as possible, perfectly useless for anything outside it domain and perfectly useful within its domain, as required by the three cardinality laws.

An complicating issue is that for several domains in reality the ontology does not provide a single view on the phenomena in reality. In for example complex sociological systems there are more then one views, each view represented by an aspect model. For example enterprise ontology provides four different graphical aspect models. The question is how these views can be represented by a single model or a coherent set of models. The DMOL representation of DEMO models is expressed in XML\(^{59}\). XML is very convenient and allows the integration of multiple trees, each tree representing a separate aspect model. For the four DEMO aspect models, this has been found to be appropriate.

### 3.4.8. C4-ness Quality Criteria related to the Cardinality Laws

The formal qualities of an ontology are expressed by the notion C4-ness. This notion applies also to models expressed using a language. C4-ness is the abbreviation of Concise and Coherent and Consistent and Comprehensive. The C4-ness criteria are internal quality criteria about the conceptualization $M$ in itself (section 3.4). In Gomez-Pérez [1995] the assessment criteria C4 are investigated and found to be mandatory for ontologies.

---

\(^{59}\) Extensible Markup Language (XML) is a very flexible and convenient markup language. Developed by the World Wide Web Consortium (W3C) and widely standardized.
3. Theoretical Foundations of Information Systems Engineering

The application of three cardinality laws with their 1:1 kind of relation produce ontologies for the language metamodel and the executing software that are isomorphic to the domain ontology. In sections 3.4.5 to 3.4.7 is shown that isomorphic ontologies offer very valuable language benefits; i) the benefits of minimized expressiveness; ii) the benefit of zero entropy; iii) elimination of ambiguity, absence of anomalies, construct overload and construct excess.

From the terms Concise, Coherent, etc, it is immediately very clear that C4-ness quality specifications are valuable for engineering. However, the questions are: i) what C4-ness precisely means; ii) what the practical consequences are of a loss of a C4-ness quality; and iii) how that loss relates to the ontology of the language metamodel and the way the cardinality laws have been applied.

Answers to these three questions will be provided for the notions of Concise, Coherent, Consistent and Comprehensive.

We assume that models, even if they are C4-ness flawed, comply to their specific language metamodel. This means that any flaws of models originate from their language metamodels. We do not assume a correct metamodel and a model that does not comply to this metamodel.

**Coherence**

i.) Coherence refers to the ‘semantic meaningfulness of the symbols and their relations from every perspective’. The term “from every perspective” refers to the possibility that models can be composed of several so-called aspect models. Each of these aspect models provides a different view and many may have a different language metamodel, with a different set of symbols and a different grammar. From all different aspect models that constitutive together a model, a coherent view must be provided.

ii.) Lack of coherence would result in loss of semantic meaningfulness from different perspectives. This would invalidate the quality that the models can be interpreted by stakeholders in an unambiguous way. This is a fatal flaw.

iii.) If in the different language metamodels different primitives are being used for the same concept, then this would cause a lack of coherence. This may happen if the three cardinality laws have not been applied properly.

**Comprehensiveness**

i.) Comprehensiveness refers to the condition that the model should encompass everything that is part of the ontology. This includes all concepts and relations of the ontology; nothing is missing.
3. Theoretical Foundations of Information Systems Engineering

**ii.)** Lack of comprehensiveness implies that models do not encompass everything that is needed for our purpose. This would result in a serious lack of the appropriateness quality. Models would be “too simple and too primitive” to be useful.

**iii.)** A lack of comprehensiveness exists if the language metamodel does not include the symbols for all of the concepts and relations of the ontology. The cardinality laws have not been applied completely.

**iv.)** Examples are programming languages that are unfit for specific purposes, such as Pascal\(^60\) to write an operating system for a computer. The original version of Pascal was intended to teach structured programming and does not provide capabilities to assess directly I/O ports and registers of a CPU, which is needed for operating systems.

**Consistency**

**i.)** Consistency refers to the absence of anomalies such as internal contradictions and conflicts.

**ii.)** Lack of consistency implies the presence of anomalies; these are model representations – or part of it - that cannot exist in reality. If these models are being used for software systems then we would see fatal bugs and crashes that cannot be fixed.

**iii.)** A lack of consistency may be caused by improper constructs in the language metamodel. These constructs relate concepts in some forbidden way. The cause of this problem is that the cardinality laws have not been applied properly.

**iv.)** Examples are so-called compiler bugs, such as notorious bugs in “C” compilers. There are dedicated tools to identify suspect coding practices and compiler bugs such as Csmith and Lint\(^61\).

**Conciseness**

**i.)** Conciseness refers to the requirement that anything that is not in the domain of the ontology should not be represented in any model. Whilst the comprehensibility quality claims that models should encompass everything within the domain, conciseness demands that everything outside the domain is excluded.

**ii.)** Lack of conciseness implies the presence of concepts and relations that are not needed. This would result in unnecessary complex models but would not immediately lead to fatal flaws.

**iii.)** A lack of conciseness can be caused by an ontology that encompasses to many concepts for our specific purpose. In this case it is not the result of improper application of the cardinality laws. A lack of conciseness may however also be

---

\(^{60}\) Pascal has been developed around 1970 by Nikolas Wirth, computer scientist, ETH Zürich, born Feb 15, 1934.

\(^{61}\) Csmith, developed at the University of Utah, School of Computing, and Lint, developed by Stephen J Johnson, are well-known tools in the ‘C’ programming community.
created by improper application of the cardinality laws, unnecessary primitives have been added to the language metamodel.

iv.) Examples of lack of conciseness are found the generic programming languages such as “C” and Pascal. Their language metamodels comprise software primitives such as “integer”, “real” and “string”, or even bits and bytes. In the first place these primitives do not correspond usually to the concepts of the ontology of interest, the cardinality laws have not been applied. In the second place there is too much design freedom, any software construct can be made outside the concepts of the ontology of interest.

It is argued here that a lack of C4-ness can be traced back to improper application of the cardinality laws. Based on this reasoning, we argue that with appropriate domain ontologies, where the cardinality laws have been applied properly, we obtain language metamodels for models that do not exhibit any lack of C4-ness.

3.5. Generic Systems Development Process for Model Driven Engineering

In this section the Generic Systems Development Process (GSDP) (sections 1.1.4; 3.5.3) is used for model instance driven software engineering (MDE) using a domain ontology and high quality ontological languages, as described briefly in section 2.2.1 and shown in figure 3.7.

![Figure 3.7](image)

**Figure 3.7** Overview of the GSDP-MDE process for information system engineering, based on Guizzardi’s framework and extension with a software engine.

We need an engineering methodology, for which we apply the GSDP (section 3.5.3). We need to make models that are directly executed by a software engine, which is model driven engineering (section 3.5.1).

The result is the generic GSDP-MDE methodology applied to software engineering (section 3.5.4) that delivers high quality IT system specifications and a software engine.
3. Theoretical Foundations of Information Systems Engineering

3.5.1. Model Driven Engineering of Information Systems

The Object Management Group (OMG) defines the model-driven architecture (MDA) or model driven engineering (MDE) of software engineering that is focused on modeling of programs, systems and business processes for a specific domain. It supports and develops model-based standards and provides a set of guidelines for the structuring of specifications, which are expressed as models. In MDE, software is typically developed in a series of model transformations. The process starts from the functional specifications and ends with the 'source code' in some formal language. A major cause of failures in applying MDE seems to be the lack of rigor in defining the distinct kinds of models, including the ignoring of the fundamental differences between functional and constructional models. In addition MDE does no state explicitly anything about the quality of the models or the modeling languages that are being used; the term model refers here to a specification $S$ expressed in language $L$.

The relation between the concepts of the domain of interest and the metamodel of $L$ are not specified either which is also a major cause of problems and failures. The question whether a phenomenon that may occur in the domain of interest can be represented by one model, or a number of models, or cannot be represented at all, is not addressed. The question if models can be constructed that can represent phenomena in our domain or not, is not addressed either by the OMG.

MDE works typically through series of model transformations but in this research there are no model transformations. The GSDP-MDE approach is domain driven design (DDD), which implies that the modeling language is derived from the domain of interest. For any MDE applications, interfaces are needed to platform-specific systems, through CORBA, dot.Net or others. In the GSDP-MDE approach the specific functions of the ActionModel and StateModel processor (section 7.3.1) are abstracted and are independent from any industry standard.

PIM is an acronym for platform independent model. It implies that in the model there exist no dependencies to a implementation specific platform or technology such as, but not excluding others, the choice of a specific CPU, the choice for network interfaces, service oriented architectures etc. In order to execute a model on some specific platform, a separate platform specific translation must be carried out. PIM is an essential notion in this research since the DEMO processor should be fully platform independent.

PSM is an acronym for Platform Specific Model (PSM) and corresponds to a view defined by a platform specific viewpoint. It describes the realization of software systems. This is completely abstracted in this research.

62 The term MDA, Model driven Architecture, is a trademark of the OMG, is also used for model driven software. We use the term MDE – Model Driven Engineering.
3. Theoretical Foundations of Information Systems Engineering

The MDE approach for software systems is typically related to other standards such as Unified Modeling Language (UML), Enterprise Distributed Object Computing (EDOC) and XML Metadata Interchange (XMI). To emphasize independence from industry standards, the use of for example UML in this research has been avoided. A second reason to avoid UML is the assumption that executable UML is not (yet) suitable for execution of our domain of interest. XML is considered to be generic and implementation-independent and is appropriate.

An important issue in MDE is Model Driven Interoperability (MDI), which refers to the interoperability of different models for different enterprises that cooperate. Interoperability issues may exist at several levels, conceptual integration, technical integration and application integration. In Enterprise Ontology there exist only DEMO models, so any interoperability problems between enterprises do not exist. In fact the construction/ destruction of transaction (section 4.3.7) is perfectly adequate to model interoperating enterprises in production chains.

DSM is an acronym for domain specific modeling. It involves the use of a modeling language that is precisely designed for a specific domain. DSM is essential in this research because we aim to develop a domain specific language for enterprise ontology with specific qualities and advantages that go beyond typical domain-specific languages. The qualities we aim for are complete expressiveness for any possible model within the domain; zero expressiveness outside the domain and zero entropy for the relation between model and real world phenomena. The main advantage is that enterprise ontology models can be executed directly, without translation to languages with lower-level abstractions. There is no (intermediate) code generation. This enables the reconstruction of a model after simulation steps that alter the model state or the model itself.

In this thesis the DSM and DDD (Domain Driven Design) is applied in most rigorous way possible, the modeling language L is precisely derived from the domain ontology using the three cardinality laws, which offers many advantages, the most important being elimination of the series of model transformations, complete expressiveness of models, the before-mentioned abstraction layers and the interoperability of (enterprise) models.

3.5.2. Model Instance Driven versus Model Driven

Enterprises exhibit a dynamic behavior with allowed states and state transitions. The states and state transitions that may occur in reality are however unpredictable for each production instance. Yet, for each production instance the state information has to be kept for further calculation of other state transitions. The only way to implement this properly is to use a specific model instance for a specific production instance. This can be described in the following example.
3. Theoretical Foundations of Information Systems Engineering

In a financial services company, such as a mortgage company with 100,000 loans, there are 100,000 model instances in a repository. Any mortgage loan may have a life cycle of up to 30 years and each may be in a different stage. The life cycle of each individual mortgage is highly unpredictable due to the close interaction with the customer. The first stage is ‘under construction’ during which the financial product has to be produced. The second stage occurs during the life of the actual loan, with payments, interests, etc. The third stage is after total payment of the initial loan; for legal reasons, the product must be kept in an information system repository. The model instance should capture at any time the operating enterprise in a truthful and appropriate way, in line with the mission statement. Each time some operation is performed on the loan contract, such as a payment or a change in interest rates, the representing model instance is retrieved from the repository, interpreted, executed, modified and saved back to the repository. The modification of the model is precisely compliant to the modification of the loan contract, in line with the cardinality laws.

The use of a model to generate a fixed compiled executable that is applicable to any production instance may be adequate for some types of applications. For the application type of the example this is far too rigid; state information would have to be stored in a relational database. Any change of the model would result in a different executable and probably also a change of the data model of the relational database and a conversion project. The possibility to link, aggregate and modify enterprise models at runtime without loss of history and conversion projects offers strong support for EIS evolution for the agile enterprise.

3.5.3. Application of the Generic Systems Development Process

For the engineering of artifacts we use the Generic Systems Development Process (GSDP) [Dietz, 2006]. The GSDP is a generic model for the design process of any kind of artifact. It is a process of a number of steps that need to be taken for the design of any system. Contrary to common (descriptive) definitions of architecture, the GSDP conceives architecture as a normative restriction of design freedom. Practically, architecture is a consistent and coherent set of design principles that embody general requirements. These principles come in addition to the (specific) requirements. The application of generic functional and constructional design principles and the need to comply with requirement specifications is especially important for software engineering to curb the undesirable programming freedom. The GSDP supports two distinctive perspectives on systems, the construction perspective, represented by white box models, and the function perspective, represented by black box models.
Figure 3.8. The Generic System Development Process (GSDP) [Dietz, 2006].

Figure 3.8 exhibits the most basic steps in a design process. The starting point is the need by some system, called the using system (US), of a supporting system, called the object system (OS). By nature, this need stems from the construction of the US, so one starts with a white-box model of the US. Then one determines the requirements for the OS in terms of the construction and operation of the US. Also by nature, these requirements are about the function and the behavior of the OS, thus in terms of a black-box model of the OS. We consider them to include the so-called non-functional requirements, regarding various performance and quality aspects.

The next basic design step is to devise specifications for the construction and operation of the OS, in terms of a white-box model of the OS. A thorough analysis of this white-box model must guarantee that building the OS is feasible, given the available technology. Reverse engineering, shown in figure 3.8, is the process of investigation of an existing artifact to identify functional specifications and for example specify these as input-output relations. Later, after design, an implementation of a working artifact is made using the US ontology, a model and some technology that should meet the functional requirements.

We also adopt the important recognition that designing is an iterative process, which is not indicated in the figure. The end result of a design process is a balanced compromise between (reasonable) requirements and (feasible) specifications. Both the functional and the constructional design step are ‘fed’ by requirements and principles.

In an MDE approach, engineering is a series of model transformations It starts ideally from a fully implementation independent constructional model of the OS. The last transformation
step results into the so-called implementation model, a platform specific model. Regarding IT systems, this model is the source code of the system.

An important advantage of applying ontology is that the function and construction of the US and the OS is brought about with C4-ness quality, eliminating ambiguity. An ontological specification that is completely implementation-independent offers the advantage that at an early stage ontological models without implementation details can be used for validation and assessment. Further in the engineering process more implementation-specific design decisions for the white box model of the OS can be made and assessed. If however the first validation step is made after all implementation decisions have been made, then a redesign is usually expensive; many design and implementation decisions may have to be changed. This is often observed in software systems when requirement specifications are incomplete or incorrect.

3.5.4. The GSDP-MDE Approach applied to Software Engineering

Referring to figure 3.8, if the OS is implemented in software then the source code is the lowest level white box model of the OS, the implementation. In the MDE approach we apply an ontology and an ontological language for which the foundations and quality criteria originate from Guizzardi [2005] are represented in figure 3.6. Fig 3.7 also represents the GSDP.

Figure 3.9 exhibits Guizzardi’s conceptual framework, slightly adapted to our purposes. Guizzardi distinguishes between a collection of real world phenomena $R$ and a collection of real world phenomena $m$, which he calls are instances of the things in $R$.

Any $m$ is composed of atomic ‘things’ from $R$. Applied to for example our purposes to investigate enterprises, $R$ is the collection of all atomic things that make up any imaginable enterprise, and every $m$ is a particular enterprise, composed of atomic actors, transactions etc. A conceptualization of an $R$ is called an ontology $C$. In our case for enterprise information systems, $C$ is enterprise ontology, as defined by the $\psi$-theory (section 3.6).

Thus, such a model consists of transaction kinds, actor roles, and the connections between them. Next, every $M$ is a conceptual model of a particular enterprise $m$, as well as an instance of $C$ (by definition). In our case, this means that every $M$ is an ontological enterprise model according to the $\psi$-theory. Guizzardi requires that the mapping from $R$ to $C$, and consequently, from every $m$ to its corresponding $M$, satisfies the qualities of ontological truthfulness and appropriateness.

Guizzardi uses the term “instance of” to relate a real world phenomenon $m$ to the atomic things from $R$ in the objective world, or model $M$ and US ontology $C$. This term is not to be confused with the term instantiation which is a relation between a concept and a type in the subjective world in the ontological parallelogram of fig. 3.4 in section 3.3.3.
Truthfulness refers to the extent to which the concepts of the ontology are able to represent phenomena in reality in a truthful way for all stakeholders [Krogstie, 2000]. The ontological appropriateness quality [Krogstie, 2000] refers to how well and useful the ontology supports understanding and shared reasoning between stakeholders. A lack of truthfulness renders a model expressed in the ontology useless. A lack of appropriateness renders a model less valuable.

Because we know that our US ontology (for example the notion of enterprise ontology according to the \( \psi \)-theory, section 3.6) has the C4-ness qualities, it is possible to design a high quality modeling language \( L \), in which ontological models can be expressed. So, \( L \) comprises the diagrams, tables, and other model representations of DEMO (section 3.6). The specification \( S \) of the ontological model of an enterprise \( M \) in \( L \) is called the DEMO model of the enterprise. For every (DEMO) model \( M \) there is one and only one specification \( S \) in \( L \). Every specification \( S \) is interpreted as one and only one (DEMO model) \( M \). In other words, \( S \) is congruent to one and only one specific DEMO model, and vice versa.

This approach deviates fundamentally from the OMG (Object Management Group) for Model Driven Engineering.

i.) The OMG states nothing about the modeling language; the language \( L \) qualities and the relations between the concepts of \( C \) and the software constructs and primitives of
L related to the domain of interest. In the GSDP-MDE approach however this is essential.

ii.) The OMG MDE approach specifies platform independent models (PIM) and platform specific models (PSM). In the GSDP-MDE approach this is not relevant, it is up to the software engineer to implement the executing software engine in the best way.

iii.) The OMG MDE approach describes series of model transformations, finally to a model than can be compiled by a generic compiler. This delivers an executable program. In the GSDP-MDE approach there are no model transformations at all, a model is interpreted and executed directly. In addition a model is executed at instance level.

3.5.5. Summary of the Benefits of the GSDP-MDE approach

The engineering effort for a specific application and domain ontology includes two parts, the design of the language L and the design and implementation in software of the software engine. This has to be only once for any domain ontology and offers the highest degree of software reuse possible.

i.) Implementation of a specific information system requires only the modeling effort because the software engine executes the model directly.

ii.) Software programming and problems with software maintenance are eliminated. A new modified information system is implemented directly by a new modified model.

iii.) The functional mismatch of information systems is addressed by the appropriateness and truthfulness qualities of the ontology C.

iv.) The possibility to simulate models directly supports early validation and helps to avoid functional mismatch. The validation results may lead to a revised model.

v.) If the model instance driven engineering approach is applied, where a model instance represents a production instance, then there may be additional benefits. Models can be modified, extended and aggregated at runtime, depending on external conditions.

These results meet the requirements of the first research question.
3.6. **Introduction to the Ψ Theory**

The applied ontology for this research on enterprise information systems (EISs) is enterprise ontology, based on the Ψ theory. Here we identify the concepts that are applied. Dietz [2006] treats this theory in more detail and depth. The Ψ theory (Greek letter Ψ or ‘psi’) is an acronym of “Performance in Social Interactions” (PSI). The Ψ theory is composed of three axioms that are all well founded in philosophy and logic, together with an additional fourth axiom. These four axioms and the distinction theorem are the foundations of enterprise ontology. The Ψ theory is a set of axioms, based on observations of the real world [Habermas, 1981; Dietz, 2006], which is a falsifiable hypothesis in the empirical sciences. This means that just one observation of untruthfulness is enough to refute the theory, as described by Popper\(^{64}\). Conversely, the domain and the borders of this theory can be simply defined by the statement: ‘there where the axioms are found to be truthful’.

It is important to realize that any ontology, by its nature to represent only specific phenomena in reality and excluding others, provides a reductionist view on reality. The warning of Lao Tse (introduction of section 3) is relevant. By its nature an ontology seems to provide a consistent representation of reality which suggests a certain degree of truthfulness and trustworthiness. It is recommended to be very wary about the appropriateness of an ontology, “does it tell me the truth, the whole truth, and nothing but the truth?” For a specific purpose an ontology may provide appropriate results, while for other purposes the ontology may fail and prove to be inappropriate. For this reason one should be always wary for the potential limitations of enterprise ontology and be aware of a potential lack of appropriateness.

### 3.6.1. The Operation Axiom

The operation axiom states that the real world contains three related domains or worlds (figure 3.10):

- **i.)** A world of actors, fulfilling actor roles, the a-world. A single actor may fulfill multiple actor roles. These actors are able to do things in the real world, c-acts (acts of coordination) and p-acts (acts of performance or production);

- **ii.)** A world of communication, the c-world, containing all of the aforementioned c-acts which constitute together the coordination between actors. The actors perform c-acts, resulting in c-facts in the c-world;

---

\(^{64}\) Sir Karl Raymond Popper, philosopher of science, 1902-1994. Founder of the empirical falsification as a criterion to separate science and non-science.
iii.) A world of all productions, the p-world, containing all p-facts, together the performance or production performed by all actors. The actors perform p-acts, resulting in the creation of p-facts in the p-world.

Figure 3.10. The operation axiom indicates the three domains or worlds of reality; the world of communication or C-world, the world of actors or A-world and the world of productions or P-world. [Dietz, 2006]

We use the terms c-fact and p-fact for ontological concepts in the c-world and p-world. In line with the Φ theory (section 1.2.3), c-facts and p-facts are the results of c-acts and p-acts. In sections 4 and further the terms cfact and pfact are being used to denote corresponding software primitives.

3.6.2. The Transaction Axiom

The operation axiom (figure 3.10) specifies actor roles. The identity of the physical actors is irrelevant, abstracted.

“An ontological transaction is a generic pattern of coordination and production, carried out by two distinct actor roles that create an original result in the process of three phases: order phase, execution phase, result phase.” [Barjis, 2009]

For each unique production there is one unique transaction. The world consists of actors performing p-acts that result in the actual production and c-acts that result in a precisely known state of the coordination between any two actors at any time. The world of phenomena can be mapped on a set of linked atomic transactions executed by actors.

The production of the coordination between two actors, the dynamic behavior, is specified by the transaction axiom. This is displayed in figure 3.12 in its most elementary form, the basic transaction pattern. Figure 3.11 gives a colorful real world example (a formal representation is given in fig. 3.12) in which an actor requests flowers from another actor.
The initiator A0 (Customer) requests (rq) a certain production to be performed by the executor (Producer) A1. The executor decides to issue a promise (pm) to the initiator, confirming that the request will be honored. This stage is known as the o-phase (ordering phase) where a negotiation about the future production takes place.

**Figure 3.11.** Illustration of the transaction axiom, the basic transaction pattern. [Dietz, 2006]

**Figure 3.12** The transaction axiom related to the operation axiom. [Dietz, 2006]
Once the actors agree about the production or performance to be carried out, the executor performs the requested task, resulting in an actual production during the e-phase (execution phase).

After the e-phase, the r-phase (result phase) occurs. The executor communicates, stating (st) that the production has been performed. The initiator then decides to accept (ac) the production. The basic transaction pattern is thus completed; the requested production has been negotiated (O-phase), executed (E-phase), delivered and accepted (R-phase).

This basic transaction pattern is the most simplified version of this axiom, however it is not suitable for real world modeling because it does not allow for transaction requests to be refused or for delivered results to be rejected. An extended version of the standard transaction pattern, where the executor may decline (dc) a request or the initiator may reject (rj) the actual production, is required; this is detailed in section 4.1.10. The standard transaction pattern is considered adequate for most operations. An even more extended version allows the actor-initiator and actor-executor to cancel earlier decisions.

It is crucial to address the question, is the transaction pattern a truthful representation of any transaction we see in the real world? The basic elementary transaction is not adequate; however it illustrates the principle of the transaction axiom. The standard transaction pattern is appropriate for most transactions we observe in the real world. It is hypothesized that the ontological truthfulness of the extended pattern with cancellations is capable of capturing any imaginable transaction in the real world. The only condition is that there must be a clear true or false of a transaction act. A ‘maybe’ is not a transaction act and furthermore cultural biases or ambiguity do not play a role. As there is no time involved, transaction state transitions are asynchronous in that they may take any time; only the sequence is relevant.

3.6.3. The Composition Axiom

The composition axiom formulates the constructional view, as opposed to the functional view, of any artifact, as defined by the TAO or \( \tau \) theory (section 1.2.4) and white box model aspects.

The composition axiom states that the production of a certain task, a p-fact, may require the prior production of 1..n underlying tasks. This introduces the use of model recursion: a tangible product may consist of many parts that must be produced and assembled. This is represented in a blueprint of the product as shown in the example of the production of a bicycle in figure 3.13. The composition axiom also specifies the sequences of p-facts, or the sequence of production of the parts. Using the bicycle example, it is clear that p-fact T1 can be produced only after the underlying p-facts (T2A, T2B, T3A, etc.) have been produced, stated (St) and accepted (Ac). The composition axiom allows derivation of the bill of
material (BOM), frequently used in the industry in the manner of a recipe or construction manual.

![Diagram of a bicycle]

**Figure 3.13** Example of the composition axiom, hierarchical decomposition of the production.

### 3.6.4. The Distinction Axiom

The fourth axiom of the Ψ theory states that there are three distinct human abilities playing a role in the operation of actors. DEMO models of enterprises consist of human actors that are able to take responsibility and commitment. These abilities are termed performa, informa and forma.
Performa is the actual production of anything that changes the material or immaterial world. This is either exposing or evoking a commitment in the c-world (coordination world) resulting in a c-fact in the c-world or producing a tangible or intangible thing, a p-fact, in the p-world (performance world). These are ontological actions as specified in the transaction axiom (figure 3.12) and these are the only actions of interest with regard to our models. If anything does not change the world in a tangible or intangible way, then it is not performa. All performa is part of ontological models.

Informa refers to meaning and consists of the creation of information and the formulating of ideas, interpretation, reasoning and deduction. All of these activities or transactions are intangible and are called infological. Informa is not ontological and is not part of any of the ontological models.

Forma consists of modifications of the form, the representation of any informa coordination or production without any modification of the meaning of the informa. Forma is not part of the ontological world. Any forma activities, transactions or productions are datalogical.

The three different human abilities are shown in figure 3.14 as the ontological, infological and datalogical layers, either for coordination or for production.

The DEMO methodology aims at the ontologies of enterprises, while discarding the informa and forma activities (the transactions or production) as being non-essential at the
ontological level. Ontological models do not show any implementation specific elements at all.

The decision to denote a certain transaction as being ontological, infological or datalogical is an important and sometimes subjective one that is made by the enterprise modeler. DEMO models can be extended with implementation-specific elements, transactions and actors that represent important and relevant implementation specific design decisions of the enterprise. Some relevance criteria have to be applied here. In information systems research, the distinction between these types of transactions is lost and hence does not play a role. There are only transactions, actors and communication and if these are found to be relevant by some criteria, they should be modeled using the fine-grained and recursive DEMO modeling approach.

3.6.5. The Organization Theorem

The organization theorem states that an enterprise is a heterogeneous system composed of three distinct homogeneous systems, as shown in the center of figure 3.14:

\(i\).

The B-enterprise (business enterprise), comprising all ontological coordination and production, encompassed by the performa, as stated in the distinction axiom.

\(ii\).

The I-enterprise (infological enterprise), comprising all infological coordination and production, as stated by the distinction axiom. The I-enterprise supports the B-enterprise.

\(iii\).

The D-enterprise (datalogical enterprise), comprising all datalogical coordination and production, as stated by the distinction axiom. The D-enterprise supports the I-enterprise.

\(iv\).

The distinction of ontological versus non-ontological is important; only the ontological transactions are essential. In implementation of enterprise engineering, the organization theorem loses its significance. It is however of prime importance for modeling the ontological essential models of an enterprise.
3.7. Introduction to Enterprise Ontology

Enterprise ontology [Dietz, 2006] is a domain ontology, an ontology for collaboration and interoperability to specify enterprises (section 1.1.1).

Figure 3.15 illustrates that organizations in the real world are highly intricate entities. The complexity of all their objects and their corresponding interactions is much greater than the cognitive capacity of our ‘small’ heads. From the investigation of an organization in its entirety, the DEMO methodology can be applied to those parts in which enterprise ontology concepts are identified. The enterprise ontology and the DEMO methodology deliver the set of four aspect models represented in a formal graphical language.

Figure 3.15. The almost unmanageable complexity of an organization illustrated on the left side and the foundation of the organization, the enterprise, on the right side [Dietz, 2006].

The actors in enterprise ontology are individual natural human actors, taking responsibility and commitment. The notion of enterprise is broad in that it encompasses many different industries, cultures and nations; enterprise ontology theory is equally applicable to a multinational corporation, as it is to the mailroom of that corporation and to the bicycle shop next door.

The purpose of enterprise ontology is, as with any ontology, to support understanding, reasoning and communication between stakeholders. A major advantage is complexity.

---

65 An organization is the complex phenomenon we perceive in the world, including all of its aspects and details. An enterprise is an abstraction of an organization provided by the theory of enterprise ontology.
reduction. Another advantage is the precisely defined set of notions of the concepts of the
ontology. Once a model is validated, by way of a consensus between stakeholders that the
model represents the enterprise in a truthful way, then steps towards implementation can be
made.

3.8. The DEMO Methodology and DEMO Models

The DEMO methodology [Dietz 2006], is based on the theory of enterprise ontology.
DEMO is a methodology, implying that the subsequent steps of the modeling process are
precisely defined and always result in a validated and verified DEMO model. DEMO
addresses exclusively the ontology of enterprises, eliminating in a very strict sense all non-
ontological or non-essential transactions, actors, coordination and information. Non-
onontological transactions are implementation, construction and operation specific and are
modeled to support ontological transactions. DEMO ignores also anything that does not
belong to the p-world66, the a-world67 or the c-world68.

DEMO defines a precisely determined relationship between the production to be delivered,
the actors (roles of enterprise members) and the state of the coordination of the enterprise.
Enterprises are intended and engineered – purposeful designed - to perform certain tasks.
Once it is clear how a production should be performed, typically by way of a product
design, business process or blueprint, then the enterprise can be designed, not the other way
around. Once the ontological DEMO models are validated and accepted, optionally after
several design and validation cycles, the enterprise can be implemented for operation.

Note that the figures, the examples and the underlying business case are taken from
Enterprise Ontology, further explanation of the examples is found there.

3.8.1. The Construction Model

The construction model (CM) is expressed in a dedicated graphical language (a legend of
the organization construction diagram is given in Appendix III. A typical example is a
library69, where a customer, actor CA01 acting as initiator, wants to become a member
(figure 3.16). There is a transaction T01, to become a member, represented by the
production of a ‘membership start’ between the aspirant member CA01 –initiator- and the

66 The p-world is the world of productions, their parts and the structure of these parts.
67 The a-world is the world of actors, the actor roles, initiator or executor, that are assigned to a
specific production of the p-world.
68 The c-world is the world of communication, between two actors, initiator and executor, about a
specific production in the p-world.
69 This example is in detail investigated in Dietz [2006].
executor A01 –executor-. Before that can happen – membership being stated - a fee for the membership must be requested and paid to A01, the executor, now acting as initiator. The executor becomes actor-initiator for this production and requests payment from actor CA02, the payer of the fee. After acceptance of the payment by the aspirant member to the executor, then the executor can deliver the membership start to the aspirant member.

**Figure 3.16.** Example of the construction model (left) with the interstriction model (right) of the “library membership” example. [Dietz, 2006]

### 3.8.2. The Process Model

The process model (PM) is calculated directly, without using any additional information, from the construction model using the transaction axiom. It contains a specification of the state space in the coordination world, a specification of the state transition space and a specification of all information items used in each coordination act. Thus, it specifies the dynamic behavior of the model (figure 3.17). Note that additional business rules can be applied as expressed in the Action Model (section 3.8.4).
3. Theoretical Foundations of Information Systems Engineering

3.8.3. The State Model

The state model is a specification of the state space and the transition space of the p-world (production world).

It specifies the state space: which facts are possible, excluding any other facts that are not possible, to occur in reality in the enterprise. The state model is composed of object classes, fact types, result types and existence laws.

The language used is WOSL (World Ontology Specification Language) [Dietz, 2005]; the legend is shown in Appendix III. It is expressed by way of an object fact diagram (OFD) and an object property list.

Figure 3.18 shows an example of the OFD and the property list {property, domain, range} for a set of properties (minimal-age, annual-fee, etc.).

The syntax of WOSL is identical to Object Role Modeling (ORM) [Halpin, 2001]. ORM is another modeling methodology, specifically applied to design and query databases on a conceptual level. It is an implementation oriented modeling technique, as opposed to the completely implementation independent ontological modeling of DEMO.
Figure 3.18. The state model with the object fact diagram and the property list, of the “library membership” example. [Dietz, 2006]

3.8.4. The Action Model

The action model is a specification of the set of rules that apply to a specific coordination state transition for a specific actor role (initiator or executor) and a specific transaction. If there is no rule specified, the transaction is performed according to the transaction coordination pattern of the process model. If there is a rule, the actor’s free choice of transaction pattern is either overridden and determined by the rule or the rule represents an advice to the actor. Such a rule can be either elementary or complicated and may depend on calculations to be performed by some external calculating machine.

The example of figure 3.19 shows that for the first transaction T01, when a request is issued, the member of class M is created for the person P. A calculation is being carried out. If the age of person P is less than the minimum age then the request for transaction T01 leads to a decline – the request cannot be carried out. If the age of person P meets the requirement of the minimal-age then the request leads to a promise for transaction T01.

Example Action model rules:
on requested T01(M) with member(new M) = P
   if age(P) < minimal_age -> decline T01(M)
   ? not -> promise T01(M)
   fi
no
on promised T01(M)
   if membership M applies for reduced fee for year Y -> request T03(M,Y)
   ? not request T02(M,Y) with standard_fee(Y)
   fi
no

**Figure** 3.19  Action model, example of two action model rules. [Dietz, 2006]

The second action model states that if a promise has been issued for transaction T01, either a request is issued for T03 or a request is issued for T02. This is an example of conditional execution selecting one of two options. The action model rules do not allow for illegal states or illegal state space transitions in the coordination space.

### 3.9. Relations between the DEMO Aspect Models

As complex entities enterprises can be viewed from different perspectives, each of which provides a different image of one and the same entity. These perspectives are called models, while the actual representation is in a corresponding diagram. In principle, the same model must be represented in several different diagrams (fig. 3.20).

The challenge is to guarantee that the aspect models provide a consistent, coherent, concise, comprehensive and essential (C4E) representation; sections 3.4.8 and 6.4 provides additional reasoning for this claim. The important aspect models are (i) the construction model, (ii) the process model, (iii) the state model and (iv) the action model.
The Construction Model
The upper part of the triangle shown in figure 3.20 is the construction model (CM), represented by the actor-transaction diagram, which is a static representation of the enterprise, composed of actors and transactions. The transactions are, if applicable, hierarchically structured according to the composition axiom. From here, the other aspect models can be derived.

The Process Model
The process model is directly derived, calculated, from the construction model, without the acquisition or application of more information, using only the transaction axiom applied to the c-acts and c-facts of the transactions. This axiom specifies precisely the static state space and the kinematic (or dynamic) state transition space of transactions. As the communication is solely represented by the c-facts, this specifies the c-world completely.

The State Model
The state model specifies the state space and the state transition space of the production world, or p-world; it is a declaration of object classes and fact types, result types and coexistence constraint rules, expressed in ORM (Appendix III).
The state model states the facts that are the result of acts in order to give a truthful representation of the enterprise to the so-called MIS type of IT systems. These systems require the facts to calculate meaningful management information to the stakeholders and its completeness and truthfulness are paramount. The fact types are defined by the requirements of the MIS IT systems. The flexibility to define all required fact types in the state model and modify this is of great practical importance.

Figure 3.20. The four ontological DEMO aspect models; CM (construction model), PM (process model), SM (state model), and AM (action model) related to each other. [Dietz, 2006]
The Action Model
The construction model, together with the process model and the state model, specify an enterprise at the ontological level completely except for any specific business rules. Business rules state that under certain conditions, a certain decision in the c-world to be taken by an actor must be taken as prescribed by the business rule.

3.9.1. Quality Assessment of Enterprise Ontology for GSDP-MDE
It must be determined whether or not enterprise ontology represents enterprises in a truthful and appropriate way and how well it supports the stakeholders in reasoning and shared understanding. Enterprise ontology has a strong ontological appropriateness, this is supported by:

i.) The theoretical foundations in philosophy and sociology, the Φ, Ψ theories and the Ψ theory with the four precisely postulated axioms and its theorem.

ii.) The DEMO methodology with the four formal graphical models that represent an enterprise from any angle with C4-ness qualities; enterprise ontology is an ontology in the formal sense. In section 6.4 supporting reasoning for this claim is provided.

iii.) The supporting empirical evidence provided by Mulder, J.B.F. [2006], Op’t Land, M. [2008], [Algoe, Boedhram, 2009] and many others and the wide acceptance by many other researchers, the community [DEMO, 2012].

iv.) The fact that currently a GSDP-MDE software production system exists in professional production.

The comprehensibility appropriateness is deemed to be sufficient but demands knowledge of the theory and the methodology, which is a normal prerequisite of understanding anything in a given language.

It is therefore considered that enterprise ontology is an appropriate ontology for the GSDP-MDE engineering of EISs. However, supporting evidence from a significant number of case studies is required.

3.10. Conclusions and Results
Software development and the functional specification quality for enterprises is often too problematic; the root cause of these problems is the paucity of an appropriate scientific theory, which manifests itself by the lack of high quality software specifications. This leads to functional mismatch or the failing business-IT alignment and the commonly encountered unmanageable two software programming problems. These problems resist the flexible adaptation required to support the agile enterprise.
3. Theoretical Foundations of Information Systems Engineering

The first research question addresses these problems and has been formulated in a generic, domain independent way since they are manifest in almost any complex information system.

The answer to the first research question is the GSDP-MDE methodology to engineer IT systems for any domain. It addresses the functional mismatch problem, the constructive specifications problem and the software implementation related problems. It is shown that substantial advantages can be achieved: the elimination of software programming and related problems, the benefits of minimized expressiveness, elimination of ambiguity, absence of anomalies, construct overload and construct excess. The highest possible degree of software reuse is obtained, the modeling language and the executing software engine have to be developed once and only once, though this may require a substantial effort.

The theory of enterprise ontology has been investigated and found to be appropriate as the ontology to be used for EISs.

These results provide an appropriate answer to the first research question. The GSDP-MDE methodology will be used to engineer the $\Psi$ processor and the DEMO processor, derived from the theory of enterprise ontology, investigated in the following sections.
PART III

The GSDP-MDE Methodology

applied to the ψ Theory
4. Design of a Static Model for the Ψ Theory

In a mathematical calculation, any error that can creep in, will creep further and it will be in the direction that will do the most damage to the calculation.

Paraphrase of Murphy’s law for computer science.
Some computer scientists even claim that Murphy is an optimist.

Abstract. In section 3 the generic GSDP-MDE methodology has been investigated. This methodology enables the construction of a model driven software engine that executes models expressed in a dedicated modeling language for a specific application domain, with elimination of programming and many programming related problems. For the application domain, a domain ontology should exist with a high degree of ontological appropriateness, truthfulness and completeness and C4-ness qualities. The systems ontology of the software engine and the modeling language should be isomorphic to the domain ontology, which can be achieved by the application of the three cardinality laws. In this section the static specifications of the systems ontology for the Ψ theory are derived and expressed in axiomatic specifications using the first cardinality law. The DMOL modeling language grammar is specified in EBNF with production rules for model construction and destruction. A near-natural language version is specified for shared reasoning.

4.1. Introduction

The subject of this section is to design a software engine for enterprise information systems (EISs) that executes enterprise ontology DEMO models using the GSDP-MDE approach (section 3.5.4). This is formulated in the second research question:

How to construct a finite state automaton, or a set of finite state automata, using the GSDP-MDE methodology, that is compliant to the Ψ theory?

The appropriateness and truthfulness qualities of enterprise ontology have been assessed and approved. For enterprise ontology the four DEMO graphical languages for the four DEMO models are already available. These languages are highly suitable for shared reasoning between humans but not for our purpose to use a computer-readable model. We need a second language for which a XML representation is chosen. This language is called DMOL (acronym of DEMO Modeling Language). According to the GSDP-MDE approach the ontologies of the two languages and the software engine have to be isomorphic to
4. Design of a Static Logic Model for the Ψ Theory

enterprise ontology. This implies that any verified DEMO model\(^70\) can be expressed in one and only one verified DMOL model vice versa. The GSDP-MDE methodology shown in fig 3.9, section 3.5.4, applies also.

![Figure 4.1](image-url) Overview of the GSDP-MDE process for enterprise ontology information system engineering, derived from Guizzardi’s framework.

The Ψ theory is the part of enterprise ontology that must be investigated first and should result in the Ψ processor. At a later stage the Ψ processor will be extended to the DEMO processor with full support for the four DEMO aspect models.

The single arrow between the DEMO aspect languages denotes that DEMO models can be entered into the DEMO processor in a convenient way using an editor. The DEMO processor does not (yet) generate the four graphical DEMO aspect models.

Models expressed in DMOL can be read, executed, modified and written back in DMOL. In this view a DMOL model is a ‘document that specifies a model’ and that is read and modified by an editor, the DEMO processor.

In this section the static specifications of the Ψ theory are investigated; the static OS ontologies (figure 3.9) of the metamodel of the DMOL language and the Ψ processor. The transaction axiom and the dynamic behavior is investigated in section 5. The first cardinality law (section 3.4) is applied:

\[ m : M : S = 1 : 1 : 1 \]  

[Cardinality law 1]

The following topics are investigated in this section:

i.) The specifications of the static Ψ models

---

\(^{70}\) The terms ‘model M’ and ‘specification S’ refer in section 3 to a conceptualization versus a specification expressed in a language. In this and the following sections the term ‘model’ is also used to denote a specification expressed in a language. From the context is clear what is meant.
4. Design of a Static Logic Model for the \( \Psi \) Theory

\( ii. \) The operation axiom and composition axiom of the \( \Psi \) theory are investigated in detail.

\( iii. \) From these axioms, a number of predicates and a tuple with sets of elements are formulated for static space.

\( iv. \) The DMOL (DEMO modeling language) representation in XML is specified.

**The construction and destruction of \( \Psi \) models**

A set of production rules, common practice in computer science, is specified to construct new verified \( \Psi \) models from existing verified \( \Psi \) models. This includes destruction of obsolete model elements. Model construction enables the construction of a \( \Psi \) model for any imaginable phenomenon in reality while the model is correctly verified and of unlimited size. The formal correctness of model construction and destruction is proven by induction.

**The rendering of DMOL \( \Psi \) specifications**

The grammar the DMOL language, represented in XML, is specified. From an existing \( \Psi \) model in the \( \Psi \) processor, the specification of that model can be rendered and any model represented in the DMOL language can be reconstructed into the original \( \Psi \) model without any loss. The extended version of DMOL grammar with the extensions of the DEMO processor is also specified. There is a near-natural language version for convenient shared reasoning between human stakeholders.

4.2. \( \Psi \) Theory derived Axioms and Propositions

The \( \Psi \) axioms are specified in schemes and in natural language. Careful analysis must be applied to identify all of the elementary or atomic true propositions in natural language, with the obvious requirement for completeness. The \( \Psi \) axioms under investigation are the operation axiom, the transaction axiom and the composition axiom (section 3.6), hereunder reformulated in 16 atomic axioms.

Based on the 16 atomic axioms, the concepts and relations of the \( \Psi \) theory are specified by a set of propositions in FOL\(^{71}\) (first order logic). FOL is a constrained descriptive language where only permitted elements and their relations are specified, which is necessary for axiomatic specification of information systems. The propositions apply to any \( \Psi \) model and should prohibit any models that do not comply with the \( \Psi \) theory.

---

\(^{71}\) First Order Logic is an elementary course in computer science, discussed in many textbooks such as: Barwise, Jon (1977); "An Introduction to First-Order Logic", in Barwise, Jon, ed. (1982). Amsterdam, NL: North-Holland. ISBN 978-0-444-86388-1.
4. Design of a Static Logic Model for the $\Psi$ Theory

Notes:

i.) The terms model and tree are equivalent; any $\Psi$ model is a tree in several aspects, as will be shown. The terms parent, child and grand-parent describe these hierarchical relations.

ii.) The term p-fact, not to be confused with the term pfact, refers in enterprise ontology to a fact about a production. The $\Phi$ theory states that Facta are things that have occurred, such as a production (a p-fact) that has been delivered, as a result of an act or acta (a p-act). In the objective domain of automaton specifications, a pfact refers to an (software) object that represents a p-fact. Similarly a cfact refers to an (software) object that represents a c-fact.

iii.) About the existence in time of p-facts or c-facts and corresponding pfacts and cfacts care must be taken. The pfact software object exists –has been modeled or programmed to be there- and is the subject of communication represented by the cfact, while the actual ontological fact, the p-fact, may not yet or may already exist as the result of a p-act. Whether a p-fact already has been produced or not, follows from investigation of the status of the software object cfact. The term cfact refers to an object with eight attributes that represents communication facts that may or may not yet have occurred.

iv.) When a DEMO model is mentioned, it means in the present section the DEMO construction model, represented by the DEMO actor transaction diagram (ATD). The term $\Psi$ model is identical in this section.

**Operation Axiom**

From the operation axiom (figure 3.10) we can formulate a derivation, comprising a set of more detailed axioms that defines sets of a tuple$^{72}$.

**Axiom 1**

There is a set of actors$^{73}$.

**Axiom 2**

There is a set of pfacts.

**Axiom 3**

There is a set of cfacts.

---

$^{72}$ A tuple is a set of sets. The elements of a set are here abstract, represented by software objects. We use in general the same terms for these software objects as for the concepts of enterprise ontology. From the context is clear what is meant.

$^{73}$ An actor is not a term used in reference to a natural person, but refers to a role to be executed by some abstract actor. There is obviously a separate mapping of natural persons to abstract actor roles. As a result, a specific natural person may fulfill multiple unique roles in a DEMO model.
Axiom 4 Each unique pfact has a relation to one unique cfact, vice versa.

Axiom 4 states that there is a 1:1 cardinality relation for a certain pfact and a cfact. In DEMO models it is possible and correct to specify a transaction with multiple initiators. This is a convenient simplification; it is allowed because in DEMO models, types are modeled. In the objective domain, we are dealing with a single instance of a type; there is one pfact modeled for each specific pfact instance and for each pfact being modeled there is one and only one cfact instance. So, in modeling on instance level, Axiom 4 is always true.

Axiom 5 Any actor may have an initiator-relation to 0, 1..n pfacts, and has one executor-relation to one specific pfact\textsuperscript{74}, except for the so-called ROOT\textsuperscript{75} initiator who responds to an external trigger instead of executing an executor role and is the initiator of the ROOT pfact and the ROOT cfact.

Axiom 6 Objects\textsuperscript{76} representing actors, pfacts and cfacts are unique instances and can be distinguished by a unique identifier, represented as an attribute ‘identifier’ related to that object.

Composition Axiom
The composition axiom states that pfacts are ordered in a hierarchical tree structure, representing the fact that for a certain pfact to be performed, first any number of child pfacts must have been performed. This is recursively the case. Note that this is constructional recursion, not functional recursion.

---

\textsuperscript{74} There is a special case where an actor is both initiator and executor for a single transaction - a self-initiating actor - at the root of the DEMO ATD model. This refers to two distinct actor roles related to the same pfact, fulfilled by the same actor, as specified in axiom 10.

\textsuperscript{75} The root of a hierarchical node tree is the one and only one node without a parent node.

\textsuperscript{76} The term object refers to software objects.
Figure 4.2. The blueprint of the production of a bicycle is an example of the application of the composition axiom.

**Axiom 7** There is a constructional decomposition type of relation between a specific pfact and any number (0, 1, ..., n) of child pfacts.

**Axiom 8** All child pfact(s) of a pfact have to be performed (stated and accepted) before the execution phase of the parent pfact can start. This complies with the Ψ theory (section 1.2.3) in a strict way. The world is composed of Acta, Facta and Stata. Acta are the cause of Facta; an act results in a fact. A pfact can be used for the construction of a parent pfact if and only if that pfact exists, has become a fact. This is true when the pfact has been stated by the executor and accepted by the initiator\(^77\). Before acceptance by the initiator the pfact does not exist and cannot be used for “anything”.

\(^77\) This strict interpretation may be subject of discussion because in DEMO modeling practice sometimes deviations are found; parent transactions being stated and accepted while child transactions are not (yet) stated and accepted. In this research we assume this strict representation of the Ψ axioms. This stance is also based on common sense and observation of the world. A part can be used for some purpose only when that part has been produced and both actors agree to this (St) and (Ac). In section 7.3 the conditional execution of transactions using the action model processor is introduced for the DEMO Processor, which allows controlled deviation from the execution sequence. The question whether this is desirable or not, cannot be answered here; it is a decision for the modeler and it is enabled.
4. Design of a Static Logic Model for the Ψ Theory

**Axiom 9**  
An actor with an executor relation to a specific pfact has an initiator relation with each child pfact of that specific pfact.

**Axiom 10**  
There is one and only one pfact, the root pfact, that has a relation to an actor that has exclusively an initiator relation\(^{78}\) to this root pfact.

**Axiom 11**  
There is at least one pfact in any model without any child pfacts – known as a terminal pfact – that has a relation to an actor that has exclusively an executor relation to this terminal pfact.

**Axiom 12**  
There are attributes uniquely related to each element; actor, cfact and pfact that describe the state of that element.

**Axiom 13**  
There are relations that relate each unique attribute to a unique element.

**Transaction Axiom**
The transaction axiom states that any transaction follows a precisely specified pattern; there are certain state transitions and rules that specify allowed and exclude forbidden state transitions.

![Diagram](image)

**Figure 4.3.** The Transaction axiom, the basic pattern, an illustration. [Dietz, 2006]

---

\(^{78}\) This is the ROOT initiator who receives and acts upon an external trigger. The ROOT initiator has no executor role.
The illustration of figure 4.3 shows the simplest form, the basic transaction. This representation involves the so-called “happy flow” of a transaction. It is assumed that the transaction always results in the delivery by the producer of the flowers.

The standard pattern (fig. 4.4) involves for each transaction step the option for the other actor to deviate from the “happy flow”. The “acts” and “facts” according to the Φ theory (section 1.2.3) are shown. After a request from the initiator, actor A0, the executor A1 has the option to ‘promise’ or to ‘decline’ the request. These acts result in the fact “Promised”(Pm) or the fact “Declined”(Dc). An act “State” by executor A1 results in the fact “Stated”. The initiator may decide to either perform the act “accept” resulting in the fact “Accepted” (Ac) or “Rejected” (Rj) etc.

![Diagram](image)

**Figure 4.4** The Transaction axiom, the standard pattern. Derived from Dietz [2006]
4. Design of a Static Logic Model for the Ψ Theory

From the transaction axiom it can be stated that a cfact is characterized by a set of eight attributes (Rq, Pm, Dc, Qt, Ac, St, Rj, Sp) for the standard transaction pattern\(^79\). From the construction axiom follows that for the Ψ transaction diagram there are dependencies; state transitions for one transaction are executed depending on the states in other transactions of a model.

**Axiom 14** Any cfact contains a set of eight binary attributes, \textit{true} or \textit{false}, together defining the current state of the transaction.

**Action Model Axioms**

So far, the formulated axioms are derived directly from the Ψ axioms. The second category of axiom is formulated by investigation of the DEMO aspect models. The DEMO action model and the DEMO interstriction model specify certain optional rules that influence the execution of the underlying state transitions. The DEMO tuple must be extended for rules, related to a certain cfact, that are to be executed for certain cfact state transitions. Also a set of relations is required for a rule and the associated cfact.

**Axiom 15** There are action model rules, each of which is uniquely related to a specific cfact.

Action model rules influence state transitions so these are part of the set of rules.

**State Model**

The state model specifies the calculation of facts that are relevant to the external world, typically MIS type of information systems. The calculation is made by execution of rules, similar to the action model rules.

**Axiom 16** There are state model rules, each of which is uniquely related to one or more a specific cfacts.

4.3. The Model Tuple and Axiomatic Specifications

In this section the Ψ model and DEMO model tuple are defined and the axiomatic specifications of the static Ψ model are specified for pfacts, cfacts and attributes.

\(^79\) The transaction pattern with cancellation patterns is specified in section 5.6.
4. Design of a Static Logic Model for the $\Psi$ Theory

4.3.1. The $\Psi$ and DEMO Tuple Specifications

Evaluation of axioms 1 to 16 yields that any DEMO model is adequately specified by a 10-tuple, a ‘decuple’, in such a way that all conceptualized entities and relations are either represented directly, or can be derived from the following tuple elements:

$$< Pf, Pcom, Cf, CP, Act, Executor, R, CR, Attribs, Attr >$$ [DEMO Tuple]

where:

- **Pf**: a set of pfacts, the production base;
- **Pcom**: a relation between pfacts;
- **Cf**: a set of cfacts, the coordination base, representing the state space;
- **CP**: a set of relations between cfacts and pfacts;
- **Act**: a set of actor roles;
- **Executor**: a set of relations between a pfact and an actor;
- **R**: a set of rules and dependencies, the rule base;
- **CR**: a set of relations between cfacts and rules;
- **Attribs**: a set of attributes contained by $Pf \cup Cf \cup Act$;
- **Attr**: a set of relations between $Pf \cup Cf \cup Act$ and Attribs.

There are three important quality criteria of this tuple:

1. The set of tuples is directly derived from the 16 axioms and is complete – any entity and relation specified by the axioms is represented in the tuple.

2. The tuple is not overloaded; any entity or relation is represented by one and only one set. The first cardinality law states that entities that are specified in the axioms should be represented only once, by one unique set of the tuple. Any set cannot be used to represent multiple entities or relations. The same applies for entities within a set.

3. The tuple is minimized, meaning that there are no redundant or interdependent sets in the tuple. The C4-ness conciseness criterion (section 6.4) states that any elements or entities that do not exist in the axioms should not exist in the tuple. More sets and relations are specified further on but these are all derived from the tuple elements.

The $\Psi$ model tuple is a subset of the DEMO tuple and is investigated first. The $\Psi$ tuple is derived exclusively from the operation, composition and transaction axioms and is specified by axioms 1 to 14. The action model and the interstriction model derived rule...
4. Design of a Static Logic Model for the $\Psi$ Theory

base with additional control of state transitions is not yet included, so the tuple elements R and CR are still omitted. Accordingly, the $\Psi$ tuple describes a static model, an ontology of endurants, the static objects of enterprise ontology. The elementary $\Psi$ tuple becomes:

$$< Pf, Pcom, Cf, CP, Act, Executor, Attrbs, Attr >$$

[\Psi \text{Tuple}]

We begin with a specification of a single $\Psi$ model. All elements of the specified tuple belong to this single $\Psi$ model; there are no elements ‘outside’ of this $\Psi$ model. For all models, the number of elements in the sets and the relations is limited. At a later stage, the operations between two separate $\Psi$ models and incomplete $\Psi$ models are discussed.

4.3.2. $\Psi$ Model Pfact Axiomatic Specifications

The tuple element $Pf$ is a set of Pfacts. $Pcom$ is a relation between pfacts that are derived by application of the composition axiom. $Pcom$ specifies that zero, one or more Pfacts may be required as part or component prior to the performance of another pfact. $Pf$ is a tree with a directed parent-child type of relation, also referred to as an aggregated pfact with a sub-component pfact.

$Pcom \subseteq Pf \times Pf$  \[ P \ 1 \]

$Pcom$ is a subset of the Cartesian product of $Pf$. The function $Pcom(p,q)$ with $p,q \in Pf$ specifies the relation between pfact $p$ and pfact $q$.

$$\neg Pcom(p,p)$$  \[ P \ 2 \]

States that there is no pfact that requires the prior performance of itself; which is a contradiction.

The following propositions apply if the number of elements, $|Pf| \geq 1$.

$Pcom(p,q) \Rightarrow \neg Pcom(q,p)$  \[ P \ 3 \]

[P3] states that any pfact that is a component of another pfact does not have that pfact as component. The relation $Pcom$ is directed.

$$Pcom(p,q) \Rightarrow \forall r \in Pf, r \neq p [\neg Pcom(r,q)]$$  \[ P \ 4 \]
4. Design of a Static Logic Model for the Ψ Theory

[P4] specifies that if \( q \) is a direct component of \( p \) then \( q \) is not also a component of another pfact \( r \). In this way the direction of the tree is defined and structures with more than one ROOT pfact in a single model are eliminated.

The following conditions \( p \not= q \not= r \) with \( p, q, r \in Pf \) apply for all following propositions.

\[
\forall p \in Pf \left[ \exists q \in Pf, q \not= p [Pcom(q, p) \lor Pcom(p, q)] \right] \tag{P 5}
\]

[P5] specifies that for each pfact there is at least one other pfact with a \( Pcom \) relation. This implies that each element of the set of Pfacts is part of a single structure completely defined by \( Pcom \).

**Recursive Relations**

A helper relation to express a grandparent type of relation between hierarchical pfacts, \( Pgcom \) is defined. It states that a certain pfact is recursively part of a part (of a part, etc…), of a {grand}parent pfact. The relation \( Pgcom \) is required to specify the root of the tree.

\[
Pgcom \subseteq Pf \times Pf \tag{P 6}
\]

The relation \( Pgcom \) is a subset of the Cartesian product of \( Pf \). The function \( Pgcom(p, q) \) with \( p, q \in Pf, p \not= q \) specifies that component \( q \) is (recursively nested) part of \( p \). The term {grand}parent–child is applicable here. The production rules to derive \( Pgcom(p, q) \) by induction for any \( p \) and \( q \) of the set Pf are:

\[
Pcom(p, q) \Rightarrow Pgcom(p, q) \tag{P 7}
\]

\[
Pgcom(p, q) \land Pcom(q, r) \Rightarrow Pgcom(p, r) \tag{P 8}
\]

Two propositions derived to calculate \( Pgcom() \) from [P 7] and [P 8]:

\[
Pgcom(p, q) \Rightarrow \neg Pgcom(q, p) \tag{P 9}
\]

\[
\neg Pgcom(p, p) \tag{P 10}
\]

Proposition [P 9] implies that the \( Pgcom() \) relation is not inverse commutative. For the root of the pfact tree is derived:
4. Design of a Static Logic Model for the $\Psi$ Theory

\[ \exists p \in Pf \left[ \forall q \in Pf, q \neq p \left[ \text{Pgcom}(p, q) \right] \right] \]  

[P 11]

Proposition [P11] states that there is one and only one pfact, called the root pfact, which has as (grand)parent $\text{Pgcom}$ relation to all other pfacts. The helper function $\text{ROOT}(p)$ is true for this pfact only.

For the pfacts without any child pfacts it is derived that:

\[ \exists p \in Pf \left[ \forall q \in Pf, q \neq p \left[ \neg \text{Pcom}(p, q) \right] \right] \]  

[P 12]

Similarly, to proposition [P 11], for [P12] there is at least one pfact that has no parent–child relation as parent to any of the other pfacts. These pfacts are called terminal pfacts and represent atomic productions.

4.3.3. $\Psi$ Model Cfact Specifications

Each cfact defines an octuple $< Rq, Pm, Dc, Qt, St, Ac, Rj, Sp >$, with one and only one element in each of the sets, each representing the Boolean value $\{\text{true, false}\}$ or $\{1,0\}$, of the state request, promise, decline, quit, state, accept, reject as specified in the transaction axiom. This tuple defines completely the state of a specific transaction. In aggregated models, with more than one cfact, the set of all cfacts defines the total state space of that model.

$\text{Cf}$ is a set of cfacts. The set $\text{Cf}$ specifies the total state space of any model.

$|\text{Cf}| = |\text{Pf}|$  

[P 13]

Proposition [P13] states that the number of elements in the sets $\text{Cf}$ and $\text{Pf}$ are equal.

$\text{CP} \subseteq \text{Cf} \times \text{Pf}$  

[P 14]

The relation $\text{CP}$ is a subset of the Cartesian product of $\text{Cf}$ and $\text{Pf}$. The function $\text{CP}(c, p)$ with $c \in \text{Cf}$, $p \in \text{Pf}$ specifies that cfact $c$ is attached to pfact $p$. 
4. Design of a Static Logic Model for the Ψ Theory

∀ p,q ∈ Pf, p ≠ q [∃ c ∈ Cf [CP(c,p) ∧ ¬CP(c,q)]] \quad \text{[P 15]}

CP is a set of relations between pfacts and cfacts in such a way that for each pfact there is one and only one cfact. If the size of the sets is 1 (one) then CP(c,p) is true.

Since Pf is a tree and for each element of Pf there is one and only one element of Cf where the relation CP is defined, the elements of Cf are also a tree.

4.3.4. Ψ Model Actor Specifications

Act is a set of actor roles. In this context only actor roles are specified; the term actor denotes an actor role here. It is possible that a single physical actor performs some or all roles in the execution of a Ψ model, but this is outside the scope of this investigation. In this specification the actor roles are unique and each actor role is composed of one or two excluding actor sub roles, initiator and executor. There is always one actor performing only an initiator role, the root initiator, while all the other actors perform an executor role, as will be shown.

\[ \text{Executor} \subseteq \text{Act} \times \text{Pf} \quad \text{[P 16]} \]

The relation Executor is a subset of the Cartesian product of Pf and Act.

\[ \text{Executor}(a, p) \text{ with } a \in \text{Act} \text{ and } p \in \text{Pf} \text{ expresses that actor } a \text{ is the executor of pfact } p. \]

\[ |\text{Act}| = |\text{Pf}| + 1 \quad \text{[P 17]} \]

Proposition [P17] states that the number of actors equals the number of pfacts + 1. This is directly verifiable and true for a model with only one pfact. Proof by induction for each additional pfact to be added to the model by the Ψ model production rules (section 4.3.7) shows that this is true for any number of pfacts in a model.

\[ \forall p \in \text{Pf} \left[ \exists a \in \text{Act} \left( \text{Executor } (a, p) \land \bigwedge_{b \in \text{Act}; b \neq a} \neg \text{Executor } (b, p) \right) \right] \quad \text{[P 18]} \]

Proposition [P 18] specifies that each specific pfact p has one and only one executor relation to one and only one specific actor a. From [P 17] it follows that there is one actor without an executor role.
The helper relation, \textit{initiator}, is defined in terms of \textit{Act} specifying the role of a specific actor in relation to a specific pfact,

\begin{equation}
\text{Initiator} \subseteq \text{Act} \times \text{Pf} \tag{P 19}
\end{equation}

The function is written: \textit{Initiator} \((a, p)\) with \(a \in \text{Act}\) and \(p \in \text{Pf}\). The relation \textit{Initiator} is not part of the tuple but is derived by the \textit{Executor} relation. The relation \textit{Initiator} is specified and calculated by induction using \(\text{P 20}\) and \(\text{P 21}\) as follows:

\begin{align*}
\text{Pcom}(p, q) \land \text{Executor}(a, p) &\Rightarrow \text{Initiator}(a, q) \tag{P 20} \\
\exists p \in \text{Pf}, a \in \text{Act} \left[ \forall q \in \text{Pf}, p \in q \left[ \text{Pcom}(p, q) \land \neg \text{Executor}(a, q) \right] \right] &\Rightarrow \text{Initiator}(a, p) \tag{P 21}
\end{align*}

Proposition \(\text{P 20}\) states that the executor of a pfact is the initiator of all 'component' or child pfacts. Since the number of required child pfacts can be \((0, 1, \ldots, n)\), a single actor can execute multiple executor roles.

Proposition \(\text{P 11}\) states that there is one and only one pfact \(p\), the root pfact, for which there is no pfact \(q\) with a relation \(\text{Pcom}(q, p)\). We must state separately that for this root pfact an initiator, the remaining actor from \(\text{P 15}\), exists.

\begin{align*}
\exists p \in \text{Pf}, a \in \text{Act} \left[ \forall q \in \text{Pf}, p \in q \left[ \text{Pcom}(p, q) \land \neg \text{Executor}(a, q) \right] \right] &\Rightarrow \text{Initiator}(a, p) \\
\neg \text{Executor}(a, q) &\Rightarrow \neg \text{Initiator}(a, q) \tag{P 21}
\end{align*}

Proposition \(\text{P 21}\) states that there is one and only one initiator of the root pfact that is not executor for any other pfact. \textit{Initiator}(a, p) \(\Rightarrow \neg \text{Executor}(a, p)\) and \textit{Executor}(a, q) \(\Rightarrow \neg \text{Initiator}(a, q)\) specify that the roles of any actor related to a specific pfact are excluding.

Note that in ATD diagrams compound actors and self-initiating actors are drawn. This is permitted and convenient because ATD diagrams specify types. At the model level, where only instances are specified, actor roles are explicitly unique.

From \(\text{P 18}\) for the calculation of the \textit{Initiator()} proposition and \(\text{P 11}\) for the root pfact follows:
4. Design of a Static Logic Model for the $\Psi$ Theory

\[ \forall p \in Pf \left[ \exists a \in Act \left[ \text{Initiator} (a, p) \land \left( \bigvee_{b \in Act; b \neq a} \neg \text{Initiator} (b, p) \right) \right] \right] \]  \hspace{1cm} \text{[P 22]}

Proposition [P22] states that for each pfact there is one and only one actor with an initiator role. The number of actors equals the number of pfacts plus the one actor specified in proposition [P21]. This is true for the elementary transaction as well as holding true after each application of the production rules (section 4.3.7), proven by induction.

From [P 22] it follows, via:

\[ \text{Executor} (a, p) \land Pcom(p, q) \land Pcom(p, r) \Rightarrow \text{Initiator} (a, q) \land \text{Initiator} (a, r) \]  \hspace{1cm} \text{[P 23]}

that:

\[ \forall p, q \in Pf, Pcom(p, q) \left[ \exists a \in Act \left[ \text{Executor} (a, p) \land \text{Initiator} (a, q) \right] \right] \]  \hspace{1cm} \text{[P 24]}

Proposition [P24] states that for any number of pfacts with a relation $Pcom$, there is one and only one specific actor that has an executor relation to the aggregated or parent pfact and an initiator relation to the sub-component or child pfact(s).

Since $Pf$ is a tree and for each element of $Pf$ there is one and only one element of $Act$ where the relation $\text{Initiator}$ is specified, the elements of $Act$ are a tree. For the root Initiator (there is one and only one) it is specified:

\[ \exists p \in Pf \left[ \forall q \in Pf, q \neq p, Pcom(p, q) \left[ \exists a \in Act \left[ \text{Initiator} (a, p) \land \left( \bigvee_{b \in Act; b \neq a} \neg \text{Initiator} (b, p) \right) \right] \right] \right] \]  \hspace{1cm} \text{[P 25]}

Proposition [P 25] is derived from [P 18] and [P 24] and states that for the root pfact, the initiator only performs the initiator role. Since this pfact has no parent, this actor cannot take an executor role.

For the terminal executor(s), of which there is at least one in each model, it is derived:
4. Design of a Static Logic Model for the $\Psi$ Theory

\[
\exists p \in Pf \left[ \forall q \in Pf, p \neq q, \neg \text{pgcom}(p, q) \left[ \exists a \in \text{Act} \left[ \text{Executor}(a, p) \land \right. \right. \left. \forall b \in \text{Act}, b \neq a \left[ \neg \text{Executor}(b, p) \right] \right] \right] \right]
\]

[P 26]

Similarly to propositions [P 12] and [P 24], [P 26] states that for any terminal pfact the executor alone takes an executor role; since this actor has no child actors there are no other executor roles.

A special case is self-initiation and self-execution. Here, an actor is both initiator and executor for the same pfact. Note that though there is only one actor, there are still two distinct roles being fulfilled.

4.3.5. Attributes

Attributes specify the details of an object, an element of $Pf$, $Cf$ and $Act$. An attribute is a tuple composed of an identifier, a type and a value. Each object with its attribute identifier is unique in an enterprise ontology model; duplicate identifier values are forbidden. The label is one of the attributes needed outside the scope of this investigation.

Attribs : a set of attributes contained by Pf, Cf and Act,
with $\text{Attribs} \subseteq \text{Ident} \cup \text{Label}$
where

Ident : a set of attributes, variables containing the identifiers of the objects $(Pf \cup Cf \cup Act)$.

There is one and only one element of Ident attached to each object. Each identifier value is unique in a tree and is required to identify a specific object instance in an unambiguous way.

Label : a set of attributes, containing a label specification of the object.
There is one and only one element of label attached to each object. The label description is not unique or unambiguous for a specific object.
4. Design of a Static Logic Model for the $\Psi$ Theory

Note:
For production versions of the $\Psi$ processor and helper variables for the implementation of DEMO model dependencies, some more attributes are needed. Examples are the attribute SelfActivation, to specify if an initiator should initiate a transaction autonomously or not, and the attribute Enabled, to specify if a transaction state change can be executed. These are to be specified in a similar way. The need or the meaning of the Label attribute is outside the scope of this section. For a cfact related to a specific pfact, the Label attribute has no relevance. For the sake of a simple and consistent specification the same set of attributes has been specified for all the sets Act, Cf and Pf.

$Attr \subseteq Pf \times Attribs \cup Cf \times Attribs \cup Attribs \times Act$ is a relation specifying which attribute is linked to which object actor, cfact or pfact.

The expression $Attr (x, a)$ with $x \in \{ Act \cup Cf \cup Pf \}$ and $a \in Attribs$ specifies that attribute $a$ is linked to object $x$.

Two operations are defined to access these attributes and the cfact tuple.

$SetVal : DM \times (Pf \cup Cf \cup Act) \times Attribs \times Val \rightarrow DM$ [P 27]

where Val is a set of values for Attribs. The function $SetVal(a, b, c)$ assigns value $c$ to attribute $b$ of object $a$. If the attribute does not yet exist it is created. The function $SetVal()$ also assigns a rule as an attribute to a cfact.

$GetVal : DM \times (Pf \cup Cf \cup Act) \times Attribs \rightarrow DM$ [P 28]

The function $GetVal(a, b)$ returns the value of attribute $b$ attached to object $a$. Typically the value is string quote delimited. In a similar way the functions $GetVal(a, b)$ and $SetVal(a, b)$ are specified for the tuple $< Rq, Pm, Dc, Qt, St, Ac, Rj, Sp >$ of Cf. with $a \in Cf$, $b \in \{ Rq, Pm, Dc, Qt, St, Ac, Rj, Sp \}$ and $c \in \{ true, false \}$. These functions are required by the production rules and the state transition functions. Similarly, the function $GetIdentifier(a, b)$ returns the string quote delimited identifier of attribute $b$ attached to object $a$.

At this stage, each set and each entity or relation from these sets have been defined, the size of the sets is defined and each element from the sets Pf, Cf, Act and Attrib has been related by one and only one relation to another element. This is also true for elements of the set Act, related to a Pf element through the relation Executor, and related to other element(s)
4. Design of a Static Logic Model for the $\Psi$ Theory

by the derived relation Initiator. There are no anomalies such as isolated elements (or groups of elements) or elements related to multiple other elements. The whole structure specified in this way is a perfect tree.

Proof by induction
The proof of these and other propositions further on in this section is done by induction over the production rules (section 4.3.7). We start with the elementary transaction consisting of a single pfact linked to a cfact with an initiator and an executor and conclude that these statements are true for this model. We apply the $\Psi$ model production rules using the \texttt{LinkPfact()} operation to link an additional elementary $\Psi$ transaction and verify that these predicates are again true. By repetitive application of the $\Psi$ model production rules, any valid $\Psi$ model can be constructed with the proof that these predicates are true; this is also the case for the \texttt{DelPfact()} operation of section 4.3.9.

4.3.6. Extended Notation Conventions and Definitions
Thus far, only specifications of a static tree, in some given state, have been specified. It is necessary to specify functions or algorithms composed of statements in order to change the state space of a given model or to create a new model from an existing one. There is a limitation in FOL (first order logic) used so far; any conditional \texttt{if p then q else r} constructs; and assignment constructs are not supported but are needed here. A notation using (pseudo) C++ or another programming language deviates from the notation used so far, is too cumbersome, too implementation-oriented and hence undesirable. An proprietary notation convention using embedded FOL for statements and assignments is used with some definitions.

A variable or attribute is a tuple consisting of a variable name, a variable type and a variable value: \texttt{<name, type, value>}.

A statement is the elementary part of a program, written in an imperative programming language. Sequential execution of a statement results in a change of the state space of the variables or attributes, or a change in the sequence of statement execution. Immediately after execution of a statement, this statement as a proposition evaluates to true.

A compound statement is a sequence of assignment statements, conditional statements and functions, recursively, statements separated by a semicolon `;`, if necessary enclosed by brackets and defined using EBNF notation as:

\texttt{Statement ::= [ [ Statement; | AssignmentStatement | ConditionalStatement | Function ]}
According to regular expression rules the symbol '|' denotes an option; the symbol pair '{' and '}' denote a repetition (0, 1, ..., n) of instances; the symbol pair '[' and ']' denote a block to be evaluated. Statements in a block are separated by a semicolon.

An assignment statement sets or re-sets the value of a variable or attribute denoted by a variable name during program execution, where a value is a sequence of bits. The assignment statement is composed of a variable followed by an assignment symbol ':=' or '→' and an expression that evaluates to a typed value. After execution of the assignment, it is equivalent to a predicate.

An expression is a syntactically correct sequence of values, variables and operands that evaluate to a value after calculation.

A function, identified by a function identifier, is a separate set of statements, executed sequentially, where parameters are being passed at the function call that act as local variables within the function body. A function is an implementation of an algorithm and it may have a type void, or evaluate to a type variable and hence return a value. A function $F(p)$ with a function body composed of multiple compound statements, which allows the use of control blocks, is represented as:

$$
F(p) : \begin{cases} 
\text{Statement} \\
\text{Statement} \\
.. 
\end{cases}
$$

The execution of a function $F(p)$ under the condition that the parameter(s) comply with a Boolean Expression $B(p)$ evaluating to true is written as:

$$
F(p : B(p)) : \begin{cases} 
\text{Statement} \\
\text{Statement} \\
.. 
\end{cases}
$$

A conditional statement is a statement to be executed depending on the result of an evaluation of a Boolean function or expression and controls the sequence of statement execution.

The symbol '⇒' means 'produces', which specifies here the if-then embedding of a Boolean function that evaluates to either true or false on the left side of the '⇒' symbol, followed on the right side of the symbol by a statement, a function or an assignment statement.
4. Design of a Static Logic Model for the $\Psi$ Theory

$B(p) \Rightarrow F(p)$

In this notation, we test for a certain value of $p$ the Boolean function $B(p)$ to be either true, in which case the function $F(p)$ is executed using the value of parameter $p$; if it is false, the function $F(p)$ is not executed. The FOL notation is followed but the distinction between a Boolean evaluation and an assignment must be noted.

A state change of a relation $R$, an assignment for a given parameter $p$, is represented as:

$B(p) \Rightarrow R \quad \text{or} \quad B(p) \Rightarrow \neg R$

This allows the specification of (programmable) algorithms using the elegant expressiveness of FOL.

4.3.7. $\Psi$ Model Production Rules

A number of special functions are defined that are required to construct new verified models from existing verified models in an atomic process. The set of functions specified are based on a previous GSDP-MDE software framework developed earlier (Appendix II). This set is found to be appropriate for production using this system.

The $\Psi$ model production rules are rules that enable the construction of a new and verified $\Psi$ model, derived from one or two existing verified $\Psi$ models. The specification of production rules is standard practice and is expressed in formal grammars. Any model that can be rendered by application of production rules, starting with a verified elementary $\Psi$ model, is a formally verified $\Psi$ model. Any models that cannot be rendered using the production rules are invalid and do not comply with the necessary grammar. Each production rule specifies all state changes in the tuple $< Pf, Pcom, Cf, CP, Act, Executor, Attrbs, Attr >$ or $< Pf, Pcom, Cf, CP, Act, Executor, R, CR, Attrbs, Attr >$ tuple instance. The resulting model after application of a production rule is called a production.

There are two directions by which it is possible to construct any new $\Psi$ model from any existing model. The first is by aggregation of new transactions into the model. The second is by destruction of parts or even the whole model. The starting point of the construction of any $\Psi$ model is the single element transaction, a verified $\Psi$ model, as shown in figure 4.4.
4. Design of a Static Logic Model for the $\Psi$ Theory

Elementary model instantiation

We require the availability of a model instantiation process, the NewTrans() function, that produces new elementary $\Psi$ transaction models. This process guarantees the uniqueness of the attributes and identifiers, in order to prevent ambiguity in the set of existing models. All elements of the tuple $< Pf, Pcom, Cf, CP, Act, Executor, Attributes, Attr >$ must be created and a default value must be assigned.

\[ \text{NewTrans}() \rightarrow \{ a, b, c, p \} \text{ with } a, b \in \text{Act}; \ p \in Pf; \ c \in \text{Cf}; \ CP(c, p); \ Executor(b, p) \]  

[P 29]

The relation $Pcom(p, p)$ resulting from $|Pf| = 1$, is default false.

This function produces for each object pfact, cfact and actor, the mandatory Attributes and their Attr relations. The SetVal() function allows value modifications. The cfact is assumed to have the tuple \{$\neg Rq, \neg Pm, \neg Dc, \neg Qt, \neg St, \neg Ac, \neg Rj, \neg Sp$\} initialized as default settings. The default set of rules $R$ attached to cfact via $CR$ is empty.

4.3.8. $\Psi$ Model Construction

Two instances of the elementary model of a transaction are required as a starting point. A new $\Psi$ model is constructed by linking two existing $\Psi$ models. Assuming the existence of two $\Psi$ models with indices 1 and 2, it is possible to define new sets composed of the sets for the two tuples:

The subscript $t$ is used to specify a set of sets:

\[ Dm_t = \{ Dm_1, Dm_2, \ldots, Dm_n \} \]
\[ Pf_t = Pf_1 \cup Pf_2 \cup Pf_n \]
\[ Cf_t = Cf_1 \cup Cf_2 \cup Cf_n \]

and so forth. Similarly:

\[ Pcom_t = Pcom_1 \cup Pcom_2 \cup \ldots \cup Pcom_n \]  

[P 30]

\[ Pcom_t = Pcom_1 \cup Pcom_2 \cup \ldots \cup Pcom_n \]

and so forth, for all sets. The operation LinkPfact is defined as:

\[ \text{LinkPfact} : Dm \times (Act \cup Pf \cup Cf) \times Attributes \rightarrow Dm \]

[P 31]
The function \texttt{LinkPfact}() is defined as:

\[
\text{LinkPfact}(b, q) := \begin{cases} \\
\forall c \in \text{Act} : \left[ \neg \text{Initiator}(t_c, q) \land \forall s \in \text{Pf} \left[ \neg \text{Executor}(t_c, s) \right] \right] \Rightarrow \text{Destroy}(c) \\
\forall r \in \text{Pf} : \left[ \neg \text{Pcom}(t_r, q) \Rightarrow \neg \text{Pcom}(t_r, q) \right] \\
\forall r \in \text{Pf} : \left[ \text{Executor}(t_r, b) \Rightarrow \text{Pcom}(t_r, q) \right] \\
\end{cases}
\]

with \( b \in \text{Act} \) and \( q \in \text{Pf} \).

This function links pfact \( q \) to actor \( b \) by creating the \texttt{Initiator}(\( b, q \)) relation. The first line states that if for the Initiator of pfact \( q \) there is no Executor role, this Initiator is destroyed. This case is shown in figures 4.4 and 4.5, in which after the \texttt{LinkPfact} operation a single actor \( c \) would remain. This is not a verified \( \Psi \) model and must be destroyed. This condition with the \texttt{Destroy}(\( c \)) operation does not occur if actor \( c \) is still executor of some pfact.

The second line negates and assigns false to any \texttt{Pcom} relation of pfact \( q \) to another pfact.

The third line creates the \texttt{Pcom} relation of pfact \( q \) to the pfact \( r \) for which the \texttt{Executor}(\( r, b \)) is true. From the created \texttt{Pcom}(\( p, q \)), the relation \texttt{Initiator}(\( b, q \)) follows according to [P 20].

The operation \texttt{LinkPfact} might have been defined also as \texttt{LinkPfact}(\( p, q \)) with \( p, q \in \text{Pf} \).

Figure 4.4 shows the situation before the \texttt{LinkPfact} operation: two separate (elementary) \( \Psi \) models.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.4.png}
\caption{Two elementary separate models of transactions.}
\end{figure}
Application of the LinkPfact operation on two separate models of figure 4.4. results in figure 4.5.

Figure 4.5. Two elementary transactions, after a link operation to form one aggregated transaction model with two transactions.

Figure 4.5 shows the situation after the LinkPfact operation; actor b takes over the initiator role that was previously linked to actor c; Initiator( b, q ) is now true and Initiator( c, q ) is now false. The resulting Ψ model on the left is verified and all propositions apply, assuming that the initial Ψ model was verified. The remaining actor c is not a verified Ψ model; it is obsolete, not shown anymore and must be destroyed.

Proof of LinkPfact operations
The proof of correctness implies that any LinkPfact operation on a verified model yields a verified Ψ model. The first step is to verify that the two elementary Ψ models are verified. The second step is to verify that for a LinkPfact operation of these elementary Ψ models, the resulting Ψ model is verified, all the propositions are still true and that any remaining anomalies are destroyed properly. The third step is to verify that if both Ψ models are aggregated, being composed of more than one pfact, the LinkPfact operation is executed in exactly the same way as for two elementary models. By induction it is verified in a simple way that any resulting aggregated Ψ model is a verified model.

4.3.9. Ψ Model Destruction
The capability to destroy software objects (representing actors, cfacts and pfacts), or even complete models is required to construct a verified Ψ models from another verified Ψ model. The “soundness” capability is the possibility to construct any imaginable Ψ model starting from any other Ψ model through a series of model linking and model destruction operations. It must be possible to execute this at runtime. The destruction of objects or models must be carried out in such a way that after destruction, all relations to the destroyed objects are set to false and all related objects are destroyed to be sure that the remaining model or models are once again verified Ψ models. The destruction of objects or models does not mean that in the real world these corresponding entities are destroyed, they
simply disappear out of the model’s scope. The operation \texttt{DelPfact} delivers a verified new model and destroys\textsuperscript{80} elements that have become disconnected and obsolete.

The elementary function \texttt{Destroy()} simply deletes the object(s) to which the parameter refers. There are two overloaded \texttt{Destroy()} functions. Overloaded functions imply here that identical operating functions can be called with different sets of parameters. The known type(s) of the passed parameter(s) eliminates any ambiguity. The first \texttt{Destroy()} function is:

\[ \texttt{Destroy}(x : [\mathit{Pf} \cup \mathit{Cf} \cup \mathit{Act}]) : \texttt{Destroy}(x) \quad [P \ 33] \]

which simply destroys any object that is either a pfact, a cfact or an actor. Any attributes linked to this object are also correctly destroyed. Application of proposition \[P \ 33\] may or may not yield a verified \(\Psi\) model after execution, hence it is forbidden to call it directly. However, this function is needed in the other overloaded function implementations.

The second \texttt{Destroy()} function:

\[ \texttt{Destroy}(a : \mathit{Act} \land \forall p \in \mathit{Pf} \left[ \neg \mathit{Executor}(a, p) \land \neg \mathit{Initiator}(a, p) \right]) : \texttt{Destroy}(a) \quad [P \ 34] \]

This function is the implementation where execution takes place under the condition that the parameter passed is an actor, which has no executor or initiator relation assigned, as in the example of figure 4.5 for actor \textit{c}. Repeated application of the production rules may yield such an isolated actor and the \[P \ 34\] \texttt{Destroy()} implementation is needed.

The operation \texttt{DelPfact} is defined:

\[ \texttt{DelPfact} : \texttt{Dm} \times (\mathit{Act} \cup \mathit{Pf} \cup \mathit{Cf}) \times \mathit{Attribs} \rightarrow \mathit{Dm} \]

The function \texttt{DelPfact()} will be defined in several different stages, each of these stages being incrementally improved to cope with different conditions. The reason for this is that verification of correctness can be done for each stage. The first stage is defined as:

\[ \texttt{DelPfact}(b, q) : \mathit{Initiator}(b, q) \rightarrow \texttt{false} \quad \text{with} \quad b \in \mathit{Act} \quad \text{and} \quad q \in \mathit{Pf} \quad [P \ 35] \]

\textsuperscript{80} Software entities, objects can be destroyed in computer memory. The corresponding objects, actors, and facts in the world of reality are not destroyed by this operation. They disappear out of scope.
This function simply sets the \texttt{Initiator}(b, q) relation between actor b and pfact q to \texttt{false}, as shown in figure 4.6, top and bottom, after operation\textsuperscript{81}.

\[
\text{DelPfact}(b, q) \text{ of } [P 35] \text{ on the model of figure } 4.6 \text{ (top).}
\]

This version of \texttt{DelPfact}(b, q) has several unacceptable shortcomings. The first is that there remains a structure without an actor initiator, which does not constitute a verified model. This structure must be destroyed properly. The second problem is that the predicate \texttt{Initiator} is derived, so the underlying relations \texttt{Pcom} and \texttt{Executor} must be modified to change the \texttt{Initiator} predicate.

The \texttt{DelPfact()} function allows destruction, whilst guaranteeing verified \( \Psi \) models after execution:

\[
\text{DelPfact} \left( \begin{align*}
p \in \texttt{Pf} \land \forall r \in \texttt{Pr} & \left[ \neg \texttt{Pcom}(p, r) \right] \end{align*} \right): \begin{align*}
\forall a \in \texttt{Act} & \left[ \neg \texttt{Executor}(a, p); \text{Destroy } (a) \right] \\
\forall c \in \texttt{Cf} & \left[ \neg \texttt{CP}(c, p); \text{Destroy } (c) \right] \\
\forall r \in \texttt{Pr} & \left[ \neg \texttt{Pcom}(r, p); \text{Destroy } (r) \right] \\
& \text{Destroy } (p)
\end{align*}
\]

\[P 36\]

\textsuperscript{81} Note that the bottom scheme of fig. 4.6 is not a verified model.
4. Design of a Static Logic Model for the \( \Psi \) Theory

Proposition [P 36] destroys a transaction of a \( \Psi \) model under the conditions that the parameter \( p \) is a pfact and that \( p \) has no \( Pcom(p,r) \) relation to any other pfact, such that the pfact to be destroyed is a terminal pfact. First the \texttt{Executor} relation is set to false and the \texttt{Executor} is destroyed. Similarly, the \texttt{CP} relation is set to false and the cfact is destroyed. Then the \( Pcom() \) relation to the parent pfact is set to false. Finally, the pfact itself is destroyed. The \texttt{Destroy(p)} implementation of the last line is the implementation according to [P 34]. The two remaining limitations of this version are that it operates only for terminal pfacts and that the potentially remaining isolated initiator problem is not solved.

\[
\begin{align*}
\forall q \in \text{Pr} [Pcom(p,q) \Rightarrow \text{DelPfact}(q)] \\
\forall a \in \text{Act} [\text{Executor}(a,p) \Rightarrow \neg \text{Executor}(a,p) ; \text{Destroy}(a)] \\
\forall c \in \text{Cf} [\text{CP}(c,p) \Rightarrow \neg \text{CP}(c,p) ; \text{Destroy}(c)] \\
\forall f \in \text{Pr} [Pcom(f,p) \Rightarrow \neg Pcom(f,p)] \\
\forall a \in \text{Act} [\text{Initiator}(a,p) \land \text{RootPfact}(p) \Rightarrow \text{Destroy}(a)] \\
\text{Destroy}(p)
\end{align*}
\]

Proposition [P 35] is the \texttt{DelPfact()} function to be called when the parameter \( p \) is either a terminal or a non-terminal pfact. The first line calls the execution of the \texttt{Destroy()} function [P 32] recursively downward through the pfact tree until the child terminal pfact(s), if any, are reached. Then the function is executed in an identical way as in [P 34]. The recursion returns and the same pfact destruction of the parent pfact is executed. The destruction is performed bottom-up. The problem of a remaining isolated initiator is also solved. If the pfact to be destroyed is the root pfact then the related initiator is destroyed also. Proposition [P 35] is the final version, guaranteeing correctness for all conditions. An example of a \texttt{LinkPfact} operation on aggregated models is displayed in figures 4.7 and 4.8.
4. Design of a Static Logic Model for the Ψ Theory

Figure 4.8. Example of two verified Ψ models before the LinkPfact operation.

Figure 4.9. Resulting Ψ model after LinkPfact(CA111, T123) operation and destruction of CA122.
4. Design of a Static Logic Model for the $\Psi$ Theory

Proof of DelPfact and LinkPfact Operations

DelPfact (and similarly LinkPfact) proof of correctness implies that any DelPfact operation yields a verified $\Psi$ model, a model for which all propositions hold. The proof of correctness for DelPfact operations consists of three steps:

i.) the verification that the destruction of a terminal transaction using proposition $[P \ 35]$ is correct; results in a new verified model;

ii.) the verification that proposition $[P \ 35]$ first recursively 'walks' through the pfact tree downward and then destroys bottom-up each terminal transaction, in a similar fashion to $[P \ 35]$ results in a new verified model;

iii.) the verification that any remaining isolated actor is destroyed properly.

4.3.10. The Copy Model function

A copy function for model (parts) is required for $\Psi$ processor based software production systems:

\[
\text{Copy}(a, q) \quad [P \ 38]
\]

The Copy() function copies pfact $q$, including of its attached executors and their child objects, to executor $a$ and constructs a transaction. The Copy() function generates a new instance of $q$.

4.3.11. Relevance of the Copy, LinkPfact and DelPfact operations

The LinkPfact and DelPfact operations are needed for two reasons. The first is theoretical, regarding the construction of correctly verified $\Psi$ models. The second reason refers to the capability to self-modifying models at runtime to capture any unpredictable changes of the production instance in the enterprise. Transactions can be constructed and destroyed at runtime, depending on the actual situation and controlled by Action Model rules (section 7.3).

4.4. DMOL representation of DEMO and $\Psi$ Models

There is the practical need to render and represent $\Psi$ and DEMO models for storage in a file repository after simulation or execution. An XML\textsuperscript{82} representation is found very convenient. The four DEMO aspect models are not suitable for this. The DMOL language

\textsuperscript{82} Extensible Markup Language (XML) is a very flexible and convenient markup language. Developed by the World Wide Web Consortium (W3C) and widely standardized.
4. Design of a Static Logic Model for the \( \Psi \) Theory

is fully compliant with the \( \Psi \) theory for \( \Psi \) Tuple. The extended DMOL version for [DEMO Tuple] supports the four DEMO aspect models completely.

The grammar of the elementary version of the \( \Psi \) modeling language is specified using EBNF, which is a way to express context-free grammars of formal languages. It is based on two sets of rules: lexical and syntactical. The lexical rules specify how the tokens of the language are assembled from an available alphabet of symbols. The set of valid tokens or vocabulary of such a language is limited. The syntactical rules specify how correct or well-formed sentences in the DMOL language are formed from tokens. The syntactical rules specify a correct sequence of the tokens. Each correct sentence in the DMOL language represents a formally correct \( \Psi \) model. EBNF allows recursive specifications, which is necessary for DMOL since the \( \Psi \) processor models are based on a hierarchical decomposition and recursion.

EBNF can be described in essence as:

i.) A set of rules or production rules;
ii.) Every rule describes the syntax of the decomposition of a specific fragment;
iii.) A model is valid and can be expressed completely by the EBNF production rules if it can be reduced to a single specific rule, with no input remaining, by repeated application of the rules.

The DMOL language is the complete and comprehensive specification of all valid models that can be constructed using the \( \Psi \) production rules. The grammar is based on XML as specified and standardized by the World Wide Web Consortium (W3C). XML is very convenient for this purpose; the theory is fully developed; there are lexical grammars and parsers; there are many tools available and it is a very important industry standard. The W3C specifies parsers that reconstruct the original model specified by the grammar in a so-called DOM, or document object model.

4.4.1. \( \Psi \) Model DMOL Rendering

A \( \Psi \) model is a hierarchical tree. A well-known and generic algorithm to ‘walk’ recursively through the model tree is specified here. The recursion follows the pfact tree through the Pcom relation. At each pfact some function \( F(p) \) must be executed. In its elementary form this algorithm [A 1] is specified as:

\[
\text{TreeWalk}(p; p \in Pf, F) : \begin{cases} \forall q \in Pf \exists Pcom(p, q) \Rightarrow \text{TreeWalk}(q, F) \\ F(p) \end{cases}
\]

or:
4. Design of a Static Logic Model for the \( \Psi \) Theory

\[
\text{TreeWalk}(p : p \in Pf, F) : \left\{ \begin{array}{l}
F(p) \\
\forall q \in Pf [ P\text{com}(p, q) \Rightarrow \text{TreeWalk}(q, F)] \\
\end{array} \right.
\]

\[\text{[A 1]}\]

The first version executes the function \( F(p) \) bottom-up; at the terminal nodes \( F(p) \) is executed first and the top node last. The second version executes top-down.

The function \text{TreeWalk()} is called with two parameters, the pfact identifier to start with – not necessarily the root pfact – and an identification of the function \( F(p) \) to be executed at each pfact. In most cases the tree walking should include other elements, cfact, executor or initiator attached to the pfact. In this case the function becomes:

\[
\text{TreeWalk}(p : p \in Pf, F) : \left\{ \begin{array}{l}
\forall q \in Pf [ P\text{com}(p, q) \Rightarrow \text{TreeWalk}(q, F)] \\
\forall c \in cf [ C\text{p}(c, p) \Rightarrow F(c)] \\
\forall a \in \text{Act} \left[ \text{Initiator}(a, p) \land \text{RootPfact}(p) \Rightarrow F(a) \right] \\
\forall a \in \text{Act} \left[ \text{Executor}(a, p) \Rightarrow F(a) \right] \\
F(p) \\
\end{array} \right.
\]

\[\text{[A 2]}\]

In this implementation the function \( F() \) is overloaded; meaning that depending on the type of the parameter(s) a different implementation exists. The \text{TreeWalk()} function is required for \( \Psi \) tree verification, in rendering a set of symbols or strings that specify models, searching for a certain element with a certain identifier.

The purpose of the function \text{Render()} is to render a complete specification of a model suitable for storage and for perfect and complete reconstruction of the original model in a file. The rendered model complies with the DMOL grammar (section 4.4.2). \text{Render()} generates a set of string elements that specify either the whole \( \Psi \) model or relevant parts of it by appending output to persistent memory, a file. In this representation XML markup symbols are used. A suitable implementation is:
4. Design of a Static Logic Model for the \( \Psi \) Theory

\[
\begin{align*}
\text{Output}(" < \text{Pfact }\!>") \\
\forall z : \text{Attribs}[\text{Attr}(p, z) \Rightarrow \text{Render}(p, z)] \\
\forall c : \text{Cf}[\text{Cp}(c, p) \Rightarrow \text{Render}(p, c)] \\
\forall a : \text{Act}[\text{Executor}(a, p) \Rightarrow \text{Render}(a, p)] \\
\forall a : \text{Act}[\text{Initiator}(a, p) \land \text{RootPfact}(p) \Rightarrow \text{Render}(a)] \\
\text{Output}(" < /\text{Pfact }\!>")
\end{align*}
\]

Where the function \( \text{Output}("\text{stringdata}"\) appends the parameter of type string to the output file. This particular implementation is applied to a pfact, however implementations of cfacts and actors are similar.

\[
\begin{align*}
\text{Output}(\text{Get Identifer}(z)) \\
\text{Output}("=") \\
\text{Output}(\text{Get Val}(p, z))
\end{align*}
\]

The notation Attribute.Identifier denotes members, the identifier of the object attribute.

\[
\begin{align*}
\text{Output}(" < \text{Cfact }\!>") \\
\forall z : \text{Attribs}[\text{Attr}(c, z) \Rightarrow \text{Render}(z)] \\
\text{Output}(" < /\text{Cfact }\!>")
\end{align*}
\]

The two actors and their roles are specified as:

\[
\begin{align*}
\text{Output}(" < \text{ActInit }\!>") \\
\forall a : \text{Act}, p : \text{Initiator}(a, p) [\text{Attr}(a, p) \Rightarrow \text{Render}(a)] \\
\text{Output}(" < /\text{ActInit }\!>")
\end{align*}
\]

and
4. Design of a Static Logic Model for the Ψ Theory

\[
\begin{align*}
\text{Output}(" & \langle \text{ActExec} >") \\
\forall z \in \text{Attrs} \ [\text{Attr}(a,z) \Rightarrow \text{Render}(z)] \\
\text{Output}(" & \langle /\text{ActExec} >")
\end{align*}
\]

\[\text{A 7}\]

The whole Ψ tree, optionally starting in the root pfact, can be rendered as a DMOL file by calling the function \text{TreeWalk}(p, \text{Render}). White space as specified by the DMOL grammar can be inserted for better readability.

4.4.2. DMOL Grammar Specifications

The DMOL grammar delivers so-called well-formed XML files. These output files follow the basic grammar of XML files and can be parsed by XML parsers. In addition, the generated XML files can be verified by DTD (document type definition) of XML schemata.

The DMOL language for Ψ processor models is simple, a tree of objects with a set of attributes; a detailed derivation is not of interest. In this elementary grammar only the following elements of the tuple [Ψ Tuple] are included:

\[
\begin{align*}
\langle \text{Pf, Pcom, Cf, CP, Act, Executor, Attribs, Attr} > \quad [\Psi \text{ Tuple}]
\end{align*}
\]

The DMOL grammar for this tuple specified in EBNF (figure 4.9).

\[
\begin{align*}
\text{DEMOModel} & ::= \langle \text{DMApp} \rangle + \text{Identifier} + \text{AppBody} + \langle /\text{DMApp} \rangle \\
\text{AppBody} & ::= \langle \text{PFactChilds} \rangle + \text{PFact} + \langle /\text{PfactChilds} \rangle \\
\text{Pfact} & ::= \langle \text{PFact} \rangle + \text{PFactBody} + \langle /\text{PFact} \rangle \\
\text{PFactBody} & ::= \{ \text{Attribute}\} + \text{CFact} + \text{ActorExecutor} + | \text{ActorInitiator} | + \langle \text{PFactChilds} \rangle + \{ \text{Pfact} + \}, \langle /\text{PFactChilds} \rangle \\
\text{CFact} & ::= \langle \text{CFact} \rangle + \{ \text{Attribute} + \}, \text{CFactState} + \langle /\text{CFact} \rangle \\
\text{CFactState} & ::= \langle \text{Requested} \rangle + \langle \text{Promised} \rangle + \langle \text{Stated} \rangle + \langle \text{Accepted} \rangle + \langle \text{Declined} \rangle + \langle \text{Quited} \rangle + \langle \text{Rejected} \rangle + \langle \text{Stopped} \rangle + \langle \text{true} \rangle \mid \langle \text{false} \rangle, + \\
\text{ActorExecutor} & ::= \langle \text{ActExec} \rangle + \{ \text{Attribute} + \}, \langle /\text{ActExec} \rangle \\
\text{ActorInitiator} & ::= \langle \text{ActInit} \rangle + \{ \text{Attribute} + \}, \langle /\text{ActInit} \rangle
\end{align*}
\]
4. Design of a Static Logic Model for the $\Psi$ Theory

Identifier ::= { Digit }, Char, ( Char | Digit )
Attribute ::= Identifier + "=", Value, " "+ ::= , Ws ,
Ws ::= (" " | ( CR + LF ))
Char ::= ("a" | "b" |.. | "z" ) | ( "A" | "B" |.. | "Z" )
Digit ::= "0" | "1" |.. | "9"
CR ::= 0D Hex
LF ::= 0A Hex

Figure 4.9 DMOL grammar for the $\Psi$ tuple.

Notes regarding EBNF representation:

• ::= denotes a definition;
• | denotes an option;
• string denotes a pair of quotes delimiting a string;
• , denotes concatenation;
• { ... } denotes an arbitrary repetition ( 0, 1, ..., n );
• In some instances an abbreviated notation for ‘Requested’, ‘Promised’, etc. is used, i.e. ‘Rq’, ‘Pm’ etc;
• There is the typical forward declaration issue; the parsing of the declarations should start bottom up.
• The optional Initiator in the declaration of PfactBody is exclusively declared for the root pfact, according to [P11] and [Axiom11].
The strings that define XML branches (PfactChild, Cfact, etc.) of the tree are chosen arbitrarily and the sequence is also arbitrarily. This degree of freedom is totally insignificant; there is one and only one tree representation for any Ψ model. Figure 4.10 displays an example of a DEMO model of two transactions and its corresponding representation in DMOL is shown in figure 4.12. The file displayed in figure 4.12 has been rendered automatically by the Ψ processor and been subject to some white space editing and removal of some less relevant branches. The specification of an XML file that meets the requirements of the DMOL language can be specified in XML schema definitions (XSD). XML tools can verify the compliance of a XML file to a XSD file.

Figure 4.10. Model with two nested transactions.

The rendered DMOL file of the model of figure 4.10:

```xml
<?xml version="1.0" encoding="windows-1252" ?>
<DMApp>DEMOApplication
<PFactChilds>
 <PFact>Tyre
   <PFactPublic prmPFactIdentifier="Tyre" prmPFactLabel="">
 </PFactPublic>
 </PFact>
 <CFactPrivate prmRequested="false" prmPromised="false"
 prmRequested="false" prmAccepted="false" prmDeclined="false" prmQuited="false"
 prmRejected="false" prmStopped="false">
 </CFactPrivate>
 <CFactPublic prmCFactIdentifier="" prmCFactLabel="">
 </CFactPublic>
 </CFact>
 <ActExec>
 <Act>
   <ActPublic prmActIdentifier="Tyre Factory" prmActLabel="">
 </ActPublic>
 </Act>
 </ActExec>
 <ActInit>
 <Act>
   <ActPublic prmActIdentifier="Tyre Customer" prmActLabel="">
 </ActPublic>
 </Act>
 </ActInit>
</DMApp>
```
4. Design of a Static Logic Model for the $\Psi$ Theory

In the actual implementation of the DMOL grammar of the $\Psi$ processor, some non-essential enhancements have been applied. There is a separation between public and private attributes for elements, which is the result of a more strict development process and the existence of an inheritance mechanism. The two representations, the one specified by the grammar and the one generated by the $\Psi$ processor, are functionally identical. The example of figure 4.11 has been generated by the $\Psi$ processor; the separation between public and private members has been eliminated and only white space has been edited. This model is, just as the DEMO aspect models, essential ontological in itself; there are no implementation details, representation data or execution instructions. There are only declarations and assignments and the nesting ordering of these through XML branches. The current state of the transactions of the model is also specified between the branches <Cfact> and </Cfact>.

4.4.3. Natural Language DMOL Rendering

Since models are to be subjected to model validation by non-expert stakeholders it is an advantage to use a model representation that is closer to natural language than the XML representation. The rendering of the (near) Natural Language DEMO Modeling Language (NLDML) aspect model, is straightforward and similar to the rendering of the DMOL file. From this NLDML aspect model the original ATD can be reconstructed easily.

```
DEMO model 'DEMOApplication' 'Test version DEMO Tree'.
Declaration of Actors.
Root Actor-Initiator 'Tyre Customer';
```
4. Design of a Static Logic Model for the \( \Psi \) Theory

Actor 'Tyre Factory';
Actor 'Rubber Factory';

Declaration of Transactions.
Transaction 'Tyre';
Transaction 'Rubber';

Declaration of Transaction aggregation.
Actor-Initiator 'Tyre Customer'
requires from:
Actor-Executor 'Tyre Factory' performing 'Tyre'
requires from:
Actor-Executor 'Rubber Factory' performing 'Rubber'
end of require;
end of require.

Declaration of Transaction states.
Actor-Initiator 'Tyre Customer' for Transaction 'Tyre' has Cfact state of:
Requested = false; Promised = false; Declined = false; Quited = false;
Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'Tyre Factory'.
Query Actor-Initiator 'Request Yes|No'.

Actor-Initiator 'Tyre Factory' for Transaction 'Rubber' has Cfact state of:
Requested = false; Promised = false; Declined = false; Quited = false;
Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'Rubber Factory'.

Figure 4.12 Example of the NLDMOL representation

The NLDMOL aspect model of figure 4.12 consists of five parts:
i.) a declaration of the actors;
ii.) a declaration of the transactions;
iii.) a declaration of transaction aggregation;
iv.) a declaration of transaction states;
v.) the set of pending actor queries, if present, for each transaction according to the specification of section 5.

4.4.4. The DMOL Grammar

The DEMO processor consists of the \( \Psi \) processor extended with the abstraction layers and implementation of the action model processor and the state model processor as specified in section 7. There are two types of rules to be executed, action model rules and rules for the calculation of facts etc. Conceptually there are two different processors, technically there is one processor to be implemented. The tuple elements \( R \) and \( CR \) are added, resulting in the DEMO tuple \([DEMO Tuple], \langle Pf, Pcom, Cf, CP, Act, Executor, R, CR, Attribs, Attr \rangle.\)
4. Design of a Static Logic Model for the Ψ Theory

The rule base
R is a set of rules that specify state transitions in the state space.

\[ CR \subseteq CF \times R \]  

\[ CR \] is a subset of the Cartesian product of \( CF \) and \( R \). The function \( CR(x, y) \) with \( x \in CF \) and \( y \in R \) specifies the relation between cfact \( x \) and rule \( y \).

\( R : DM \times Event \rightarrow DM \)

specifies a set of rules \( R \) that define the state transition of the elements of the set \( DM \) to another element of the set \( DM \) depending on the elements of the set of events \( Event \). The DMOL grammar with action model rules and State model fact calculations for the [DEMO Tuple] specified in EBNF is given in figure 4.13.

The DMOL grammar for the complete DEMO models:

\[
\begin{align*}
    \text{DEMOModel} & \ ::= \ "<DMApp>" + \text{Identifier} + \text{AppBody} + "</DMApp>" \\
    \text{AppBody} & \ ::= \ "<PFactChilds>" + \text{PFact} + "</PfactChilds>" \\
    \text{PFact} & \ ::= \ "<PFact>" + \text{PFactBody} + "</PFact>" \\
    \text{PFactBody} & \ ::= \ \{ \text{Attribute} \} + \text{CFact} + \text{ActorExecutor} \\
    \text{CFact} & \ ::= \ "<CFact>" + \{ \text{Attribute} \}, \text{CFactState} | \text{ActionRules} | "</CFact>" \\
    \text{CFactState} & \ ::= \ "Requested" + "=\" + "true" | "false", " + "Promised" + "=\" + "true" | "false", " + "Stated" + "=\" + "true" | "false", " + "Accepted" + "=\" + "true" | "false", " + "Declined" + "=\" + "true" | "false", " + "Quited" + "=\" + "true" | "false", " + "Rejected" + "=\" + "true" | "false", " + "Stopped" + "=\" + "true" | "false", " + "No" \\
    \text{Rules} & \ ::= \ "<Rules>" + \{ \text{Rule} \} + "</Rules>" \\
    \text{Rule} & \ ::= \ "OnDefault" | "OnRequested" | "OnDeclined" | "OnPromised" | "OnQuitPromised" | "OnQuitDeclined" | "OnStated" | "OnRejected" | "OnQuitStated" | "OnStopped" | "OnQuitRejected" | "OnAccepted" + \{ \text{Line} \} + "No" \\
    \text{ActorExecutor} & \ ::= \ "<ActExec>" + \{ \text{Attribute} \}, "</ActExec>"
\end{align*}
\]
4. Design of a Static Logic Model for the $\Psi$ Theory

ActorInitiator ::= "<ActInit>" + { Attribute + }, "</ActInit>

Identifier ::= { Digit }, Char, { Char | Digit }

Attribute ::= Identifier + "=" + ", Value, 

Value ::= Char | Digit, { Char | Digit }

+ ::= , Ws ,

Ws ::= { " " | ( CR + LF )

Char ::= ("a" | "b"| .. | "z" ) | ( "A"| "B" |.. | "Z" )

Digit ::= "0" | "1"| .. | "9"

CR ::= 0D Hex

LF ::= 0A Hex

Figure 4.13 DMOL grammar for the four DEMO aspect models.

The Rule clauses with the element(s) Line, which represents the action model rule(s) or state model fact calculation(s), are defined and investigated in detail in section 5.5.9. The elements <Rules> </Rules> relate any enclosed action model rules or fact calculations to a specific transaction or Cfact, defined by $\mathbf{CR} \subseteq \mathbf{Cf} \times \mathbf{R}$.

4.5. Conclusions and Results

The GSDP-MDE approach of section 3 for static ontologies has been applied to the $\Psi$ theory ontology. By applying the first cardinality law, an n-tuple and a set of axiomatic specifications of objects, attributes and relations has been devised that is isomorphic to the $\Psi$ theory ontology.

Production rules are formulated for the construction and destruction of verified $\Psi$ models. The DMOL language, represented in XML and specified in EBNF, has been designed to represent $\Psi$ models. An extension of DMOL for DEMO models that include DEMO action model rules and the rules to calculate facts of the DEMO state model, has been designed. The grammar and the production rules enable the construction and destruction of any DEMO models, as well as the rendering of models in DMOL. The representation in DMOL of DEMO models is complete; any DMOL model can be reconstructed into the original DEMO aspect models without loss. The results of this section are part of the programming specifications of the software $\Psi$ processor and DEMO processor.
5. Design of a Dynamic Logic Model for the Ψ Theory

Once you open a can of worms, the only way you can recan them is to use a larger can.

Zymurgy's first law of evolving system dynamics

This Zymurgy and his first law, also formulated as "Old worms never die; they just worm their way into larger cans", should be defeated or curbed in software engineering. Larger cans refer to increasing entropy over time. This is addressed by the application of the GSDP-MDE methodology to construct model driven software engineering systems with zero entropy.

Abstract. A Ψ model is, as stated before, any model that complies with the axioms of the Ψ theory. The static Ψ model, derived following the GSDP-MDE approach (section 3.5) and using the cardinality law 1 (section 3.4), forms the basis of specifying the Ψ model dynamic or kinematic behavior. This is investigated using the second and third cardinality laws. The cardinality laws are applied to the states of the communication acts and facts, specified by the transaction axiom.

5.1. Introduction

In the previous section the first cardinality law (section 3.4) has been applied according to the GSDP-MDE approach, to the Ψ theory static ontology of perdurants.

\[ m : M : S = 1 : 1 : 1 \]  \[\text{[Cardinality law 1]}\]

The results are the specifications of the DEMO and the Ψ tuples, the production rules for models and the DMOL language metamodel. For an ontology of endurants, following the GSDP-MDE approach, the state space and transition space cardinality laws (section 3.4) have to be applied also:

\[ m[i] : M[i] : S[i] = 1:1:1; \text{ for any } i:1,..,n; \ n \geq 1 \]  \[\text{[Cardinality law 2]}\]

83 Zymurgy, a further unknown scientist and philosopher, also postulated the law that states that people are always available for work in the past tense.
and

$$T(m[i], m[j]) = T(M[i], M[j]) = T(S[i], S[j]) \text{ for any } i, j : 1, \ldots, n; \ i \neq j; \ n \geq 1$$

(Cardinality law 3)

The specifications for the static $\Psi$ models of section 4 still apply. The dynamic behavior is
represented by different possible states of cfacts. Each cfact contains an octuple $< Rq, Pm, Dc, Qt, St, Ac, Rj, Sp >$, (section 4.3.3) with one and only one element in each of the sets,
each representing the Boolean value $\{true, false\}$ or $\{1,0\}$. These represent the state of
request, promise, decline, quit, state, accept, reject as specified in the transaction axiom
(section 3.6.2). This tuple defines completely the state of a specific transaction. In
aggregated models, with more than one cfact, the set of all cfacts defines the total state
space of that model. To calculate the state space and transition space for each cfact the
cardinality laws have to be applied to the transaction axiom.

5.2. The Standard Transaction Pattern

The elementary or single level DEMO construction model, which is identical to a $\Psi$ model,
is shown in figure 5.1. The term elementary refers to the fact that a single transaction is
being represented. The term standard refers to the standard transaction pattern, as opposed
to the basic transaction pattern or the transaction pattern with cancellations.

Figure 5.1 shows a $\Psi$ model or DEMO Construction Model (interaction) expressed in the
Actor Transaction Diagram (ATD) (section 3.8.1, figure 3.16), which is a
conceptualization.

Figure 3.6 shows the first cardinality law between model elements being represented by
language primitives and vice versa in language primitives being interpreted as model

---

5 Design of a Dynamic Logic Model for the $\Psi$ Theory
elements. It should be noted that the terms pfact and cfact may refer to either software objects or model elements; from the context it is clear what is meant.

A representation of the DEMO process model derived from Dietz [2006], applied to figure 5.1 is shown in figure 5.2.

The process model is derived by application of the transaction axiom. This representation is investigated here within the domain of software objects, using terms and notions from software engineering and using binary values for attributes that represent transaction states.

Figure 5.2. The transaction axiom for the elementary construction model (right).

Notes:

i.) In the domain of the $\Psi$ processor executing a $\Psi$ model, a transaction is a transaction instance, as opposed to within DEMO modeling where transactions are represented as transaction types. The relation between instances and types for DEMO models is described in section 3.3.3.
ii.) The term *construction model*, usually graphically represented, is here equivalent to *Ψ model*. The Ψ model is usually represented in the DMOL language.

iii.) The transition state changes are discrete and asynchronous. In theory there are no time-outs. In practice, in production environments, timeouts are usually needed to recover from a frozen state due to malfunction of peripheral systems. This topic is discussed in section 7.3.1.

Figure 5.2 displays the transaction process of a transaction T1 between an actor A0, with the role of initiator, and actor A1 that performs the role of executor. Actors A0 and A1 are imperatively bound by the rules of the transaction axiom. At the decision element ‘a. Q’ shown in figure 5.2, initiator A0 is informed that a Dc (decline) has been issued by executor A1 and is offered the two valid options either to issue a Qt (quit) or a repeated Rq (request). Actor A0 has no other valid option. The scenario is similar in the cases of the other controlled and enforced decision elements B to E. There exists only one option for decision element C, which is to issue a St (state) at the time the actual production P has become available.

The communication pattern, which constitutes a communication protocol in the software domain, involves eight binary flags\(^{84}\) which can be set (value is true, or 1) or reset (value is false, or 0). The flags are: quit (Qt), decline (Dc), request (Rq), promise (Pm), accept (Ac), state (St), reject (Rj) and stop (Sp). These eight binary flags have 256 permutations (\(2^8\)). Some permutations are allowed and others are prohibited. Based on the scheme of figure 5.2 the transaction commences with all parameters reset to false, i.e. \(Rq = 0; Pm = 0; Dc = 0; Qt = 0; St = 0; Ac = 0; Rj = 0; Sp = 0\). The notation \(Tx.Rq, Tx.Pm\), etc. is used to denote the request or promise of transaction Tx for trees with more than one transaction.

There are two distinct phases in a transaction process. The first is the order phase, in which the initiator and the executor negotiate about the pfact to be executed. The second phase is the result phase, that starts when a pfact has been promised (\(Pm = true\)). During the order phase, all of the result phase parameters remain reset (\(T1.St = 0; T1.Ac = 0; T1.Rj = 0; T1.Sp = 0\)).

**The Order Phase**

All parameters for root initiator A0 (where the term root refers to the top element in a hierarchical tree) are reset during the order phase. The system (e.g. the role based control system) queries the root initiator, issuing a request to execute T1. Note that it is exclusively in the case of a root initiator that the condition \(Rq = 0, Pm = 0, Dc = 0, Qt = 0\) leads to the issuing of a request query. This is equivalent to a self-initiating actor. Since this is an asynchronous system, the root initiator or any initiator and executor may delay this request.

\(^{84}\) These binary flags are symbols that represent a communication fact, as the result of a communication act.
When the root initiator issues a request, parameter Rq of transaction T1 is set; i.e. $T1.Rq := 1$. The system then issues a question to executor A1; in this way a request has been issued and the system offers the options to choose a promise or a decline.

In case of a decline by executor A1, the parameter Dc of transaction T1 is set, $T1.Dc := 1$, and the previous request is reset, $T1.Rq := 0$. The system then informs the root initiator A0 that a decline has been issued and questions for a renewed request or a quit. If the request is repeated the whole process repeats; $T1.Rq := 1$. This is a minor loop.

In case of a quit the parameter $T1.Qt$ is set, $T1.Qt := 1$, while $T1.Dc$ remains set. As is the case for complex concatenated transactions (section 4.3.1), the $T1.Dc = 1$ must be maintained. If the initiator issues a quit, $T1.Qt = 1$, the executor must be informed and acknowledge this condition before the whole order process can be terminated. The condition $T1.Dc = 1$, $T1.Qt = 1$, leads to a question to the initiator to acknowledge the quit. This acknowledgement is an extension that is required for systems with aggregated transactions. After receipt of this acknowledgement the whole order process is terminated and all parameters are reset to the default state: $T1.Rq = 0; T1.Pm = 0; T1.Dc = 0; T1.Qt = 0$. From this state the transaction may continue.

In case of a promise, if there are no child transactions to be executed, the executor receives a question as to whether or not the creation of the pfact has been completed and a state can be issued. The order phase is then terminated and the result phase begins.

The Result Phase
The executor starts the creation of the pfact immediately after having issued a promise (if there are no child transactions as in this case). When the creation of the pfact has been performed by the executor, the executor issues a state; the parameter state is set, $T1.St := 1$. The order parameter promise, $T1.Pm = 1$, remains set; the transaction has been promised and stated but not yet accepted. The system informs the initiator A0 that the execution of the transaction has been stated and issues the question for an accept or a reject of the pfact.

In the case of a reject from the initiator, the transaction result – production has been executed and delivered but the result is not accepted. The parameter reject is set, $T1.Rj = 1$, while the parameter state, $T1.St$, remains set. The executor then informs the initiator that the transaction result was withdrawn, obviously because the transaction result could not be adapted. The initiator acknowledges this and the system returns to the initial condition. The system informs executor A1 that the execution of the transaction has been rejected and issues the question for a (repeated) state or a stop. In case of a repeated state from the executor, the system made a minor loop. Initiator A0 receives a repeated question to accept or to reject.
In case of a stop, the execution of the transaction is terminated. The executor states that the transaction cannot be completed. The order phase parameter promise is reset and as such the promise is withdrawn by the executor. This results in all negotiating phase parameters being reset, returning to the default situation: \( Rq = 0; Pm = 0; Dc = 0; Qt = 0 \). The executor also resets the reject parameter and sets the stop parameter: \( St = 0; Ac = 0; Rj = 0; Sp = 1 \), to inform the initiator about the stop. The initiator then receives a question to acknowledge the stop. After the acknowledge (Ack) from the initiator, the stop parameter is reset resulting in \( St = 0; Ac = 0; Rj = 0; Sp = 0 \). All transaction parameters are reset. The transaction state is again in the default state.

In case of an accept from the initiator parameter accept is set, \( T1.Ac := 1 \), and stated; the stop, \( T1.Sp := 0 \) is reset. Then the system resets \( T1.Pm := 1 \), with the result that all order phase parameters are being reset. The only parameter set is \( T1.Ac = 1 \), indicating that the transaction has been executed, stated and accepted.
5.3. **Aggregated Standard \( \Psi \) Transaction Patterns**

To investigate complex aggregated transactions, trees with more than one transaction, the elementary or single level model must be extended according to the composition axiom. The term standard refers to the standard transaction pattern of the transaction axiom. The first step is to consider the situation in which the production of a pfact depends upon the separate production of a single child pfact, which is a part of the first pfact; this production of the child pfact is required before the overall product can be produced. There are two production rules for decomposition.

**Transaction Concatenation**

A single transaction is split into two sequential transactions, resulting in concatenated parent – child transactions. This situation occurs if a part (construction decomposition, composition axiom) of the pfact has to be produced by another executor. The first executor takes the role of initiator for this partial pfact. The execution of the transaction is ‘bottom-up’ because the result of the child transaction is required to be ready, accepted, in order to commence the execution of the parent transaction. This situation is displayed in figure 5.3.

**The AND Parent-child Transaction Aggregation**

A single transaction is split into two transactions, where the part of that transaction that has been separated is assigned to a new executor. There is a ‘parent with two children’ structure. This situation is shown in figure 5.5 for transaction T1 with two child transactions T2A and T2B. Any number of child transactions is possible.

**Example**

A customer has ordered the production of a car\(^85\). At production time, it becomes clear that one minor part, such as a mirror, cannot be delivered. The upward rollback mechanism through all higher transactions – from mirror to door, to bodywork, to car – results in a decline of the car’s production. This complies with observations of scenarios that occur in the real world. At the same time, all other transactions for the production of other minor parts should also rollback. Finally, the pending request for the rear light must rollback because the overall production of the car and any parts of it is not needed anymore.

---

\(^85\) The purpose of this example is to illustrate the mechanisms of rollback. This situation will not likely occur at a car manufacturer because usually there will be readily manufactured cars in stock. Also an order for a car will not be declined because of a missing minor part.
5.3.1. Concatenation of Transactions

Figure 5.3 shows a transaction tree, transaction T1 with an enclosed transaction T2.

![Figure 5.3 Construction Model with transaction encapsulation.](image)

Two transactions T1 and T2 need to be executed (figure 5.3), where transaction T2 must be completed – accepted – before the execution phase of transaction T1 can commence.

This requirement applies because the result of transaction T2 is probably (but not necessarily always) part of or needed for transaction T1. Note that actor A1 plays both an executor role for transaction T1 and an initiator role for transaction T2, while the roles are interrelated. There is a parent-child type of relationship between actor A0 and actor A1 and between actor A1 and actor A2. There is also a parent-child type of relationship between transaction T1 and transaction T2. The terms subordinate, parent and child refer to this type of relationship. In this case, there is only a single child transaction T2. Any number of child transactions is allowed. The DEMO process model of the construction model of figure 5.3 is shown in figure 5.4.
Figure 5.4 Process model of figure 5.3, transaction encapsulation mode, where mode is termed ‘promise after request’ or PaR.

Note:
The executor issues a promise whilst assuming that all subordinate transactions can be executed by the subordinate actors. If this is not the case, a subordinate actor will issue a decline upon realizing this fact; then the executor can revoke the earlier promise and issue a decline. This is in line with the transaction axiom. Not all transaction state transitions are specified here in detail; the transitions are specified in the axiomatic specifications, section 5.5.
5 Design of a Dynamic Logic Model for the Ψ Theory

5.3.2. The AND Parent-Child Transaction Aggregation

The model of a system with multiple child transactions, the n-ary relation with n>1, is shown in figure 5.5. Before transaction T1 can be executed, the transactions T2A and T2B must have been performed and accepted. Similarly, these two transactions T2A and T2B require the prior execution and accept of transactions T3A and T3B. There is an ‘AND’ type of relationship between transactions T2A and T2B; both transactions must be accepted before the parent actor A1 can execute the production of transaction T1.

![Construction Model with multiple encapsulated transactions trees.](image)

Figure 5.5. Construction Model with multiple encapsulated transactions trees.

The transaction patterns T2A and T2B and their interactions are of interest when considering an unlimited chain of grandparent-parent-child transactions; although this is out of the scope of the current discussion, it demonstrates the potential of adopting such a hierarchy.
Figure 5.6. Process Model derived from the construction model of figure. 5.5.
Transaction patterns with child transaction cardinality of more than two are similar to the cardinality of two shown in figure 5.5. This pattern serves as a production rule that to be applied in order to build unlimited large aggregated models from the elementary model. The process model of figure 5.5 is shown in figure 5.6.

5.4. Transmission and reflection Phenomena

In state transitions of aggregated Ψ models, several phenomena emerge, an example of which is examined here. In most aggregated Ψ systems state several propagation phenomena occur: one ‘upward’ – towards parent transactions and actors – and several ‘downward’ – towards child transactions and actors.

Figure 5.7 Construction Model of three concatenated tasks.

This is illustrated in the simple transaction concatenation shown in figure 5.7, whilst more detailed investigations of these rollback and propagation phenomena are provided in section 6.2 and figure 6.5.

In this example, only the order phase with regard to transaction T1 is investigated. Before production of transaction T1, it is necessary to complete the production of transactions T2A and T3A such that they are accepted. To demonstrate these phenomena, a time axis is introduced (figure 5.8).
Figure 5.8 State transitions example in the time domain with asynchronous time intervals.

Figure 5.8 shows a transaction pattern for which only the order phase is investigated. At time = t1 a request is issued by actor A0 to actor A1. At t2 actor A1 issues a promise to actor A0 and a request to actor A2A. At t3 actor A2A issues a promise to actor A1 and a request to actor A3A. Actor A3A refuses for some reason (the reason being irrelevant to this example) and issues a decline to actor A2A t4. Actor A2A decides, again for some reason, to repeat the request at t5. At t6, actor A3A repeats the decline. This cycle t3-t4-t5-t6, a minor cycle, can be repeated ad infinitum.

At t7, actor A2A decides to quit this cycle. Initiating actor A2A issues a quit to actor A3A and executing actor A3A withdraws his earlier promise, replacing it with a decline by setting Pm = 0 and Dc = 1 for transaction T2A. Actor A1 recognizes the state transition of transaction A2A and decides to repeat his earlier promise by resetting decline and setting request for transaction T2A at t8.
At t9, actor A2A issues a promise for transaction T2A, issuing a request again for transaction T3A and also removing his earlier pending quit (Qt = 0) for transaction T3A. This quit has obviously not yet been handled (meaning in this case that it has not been acknowledged) by actor A3A. The ability to issue a command, in this case Qt = 1, and to later withdraw this command, in this case by resetting the Qt = 0 and issuing another command Rq = 1, is required and allowed.

Actor A3A declines again by setting Dc = 1 at t10. Actor A2A issues a quit to actor A3A and a decline to actor A1 at t11. Here one state transition results in two propagation waves; the downward moving quit wave and the upward moving decline wave. Much later, after t13, actor A3A acknowledges this quit by resetting Qt = 0. Actor A1 handles the decline from actor A2A at t12 by issuing a decline to actor A0 and a quit to actor A2A. At t13 actor A0 handles the decline and issues a quit to actor A1. However, at this stage actor A0 might decide to repeat his request that was first issued at t1. In this case there is a major cycle or loop of request and decline over multiple transaction levels.

Major and minor cycles may occur for request-decline negotiations and for state-reject negotiations. Collisions may occur when for example an upward moving wave such as a decline meets a downward moving wave such as a request or a quit. The detailed state transitions discussed further in this section handle these conditions. Several other minor and major cycles are possible. Minor cycles occur within a single transaction. Major loops span multiple transactions.

### 5.4.1. Transaction Axiom Extensions for aggregated Transactions

The transaction axiom specifies a single transaction. An extension, not a modification, is necessary for aggregated transaction trees. This is because for a single transaction a final state can be reached, while for any transaction tree the whole process may not yet have reached a final state. In other words, while for a single transaction a final state may have been achieved, the other transactions of the tree are not yet in a final state.

For a standard \(\Psi\) transaction, if the executor issues a stop or if the initiator issues a quit, the transaction reaches a final state. In any transaction tree this would imply that all other transactions are ‘ignorant’ of this condition and the whole transaction system would freeze. To rectify this, two extensions are required to comply with real world situations:

The act of an acknowledgement of a quit or a stop. In case of a quit from the initiator, the executor should acknowledge this by issuing a quit acknowledge. In case of a stop from the executor, the initiator should acknowledge this by issuing a stop acknowledge. Expressly,
the system informs the executor and the initiator, issuing a question for an acknowledgement.

After the quit acknowledge or stop acknowledge any other transactions where this actor participates, either as executor or as initiator, are also affected. The effect of a quit or a stop propagates by one of the two rollback mechanisms. This is implemented by additional state transition axioms in such a manner that for every transaction the actor roles comply with the transaction axiom.

5.5. State and Transition Space Specifications

The transition space functions are directly derived from models such as that displayed in figure 5.6, in which the 8 elementary parameters are a point in an eight-dimensional state space for each transaction. The state and transition space specifications are algorithms; hence if certain conditions apply, a state transition occurs. For the state space and transition spaces a vector representation is considered to be the most appropriate and convenient programming specification. The standard transaction pattern and the additional transaction cancellation patterns are specified in this manner.

5.5.1. Notation Conventions

Cfacts, software objects, are defined by the state at any particular point in time of the 8 elementary parameters, \{Rq, Pm, Dc, Qt, St, Rj, Ac, Sp\}, having values \textbf{true} or \textbf{false} (1 or 0), shown in the following example:

\[
\begin{pmatrix}
\neg Rq \\
Pm \\
Dc \\
\neg Qt \\
St \\
\neg Ac \\
Rj \\
\neg Sp
\end{pmatrix}
\]

This notation is equivalent to: \(\neg Rq \land Pm \land Dc \land \neg Qt \land St \land \neg Ac \land Rj \land \neg Sp\)

We need to specify functions in order to conditionally change the current state in the state space. This involves a Boolean evaluation for a conditional state change. The limitation in FOL, of evaluation and conditional assignment constructs ‘if p then q else r’ being
unsupported, is solved in a pragmatic way. Some (pseudo) C++ representations are defined here to overcome these limitations.

The symbol ‘⇒’ means ‘produces’ and specifies here the conditional if-then embedding of a Boolean function. The Boolean evaluates to either true or false on the left side of the symbol, followed on the right side of the symbol by a (compound) statement, a function or an assignment statement.

\[ B(p) \Rightarrow F(q) \]

In this notation we test, for a certain value or values of parameter (set) \( p \), the Boolean function \( B(p) \) to be either true, in which case the function \( F(q) \) is executed using the value of parameter(set) \( q \), or false, in which case the function \( F(q) \) is not executed. The FOL notation is followed; however, the distinction between a Boolean evaluation and an assignment must be noted. An arbitrary example is given here:

\[
\begin{align*}
\text{Actor.Exec} & \quad \Rightarrow \quad \text{Actor.Init[i]} \\
\begin{cases}
Rq = 0 \\
Pm = 1 \\
Dc = 1 \\
Qt = 0 \\
St = 1 \\
Ac = 0 \\
Rj = 1 \\
Sp = 0
\end{cases} & \quad \Rightarrow & \quad \begin{cases}
Rq := 0 \\
Pm := 1 \\
Dc := 0 \\
Qt := 0 \\
St := 1 \\
Ac := 0 \\
Rj := 1 \\
Sp := 0
\end{cases}
\end{align*}
\]

The symbol ‘==’ represents a (C++ like) Boolean evaluation; 0 represents false and 1 represents true. The Boolean \( Rq == 0 \) evaluates to true if \( Rq \) is false. The Boolean state conditions shown in the example in the left side of the symbol \( \Rightarrow \) is equivalent to the previous example:

\[ \neg Rq \land Pm \land Dc \land \neg Qt \land St \land \neg Ac \land \neg Rj \land \neg Sp \]

The term Actor.Exec( ) denotes a specific actor executor role of an actor. The notation actor.Init[i]( ) denotes an actor initiator role, where the index \( i \) identifies the specific initiator role. ‘\( i \)’ is the set of all initiator roles for a specific actor; \( i \in I \). If all vector elements evaluate to true then the right side of \( \Rightarrow \) is assigned and results in:

\[ \neg Rq \land Pm \land \neg Dc \land \neg Qt \land St \land \neg Ac \land \neg Rj \land Sp \]
The symbol "\:=\" represents an assignment; immediately after the assignment $a:=b$ the Boolean evaluation $a=b$ evaluates to true. The term $Pm,Dc:=1$ is equivalent to $Pm=1 \land Dc=1$ and the assignment notation $Dc:=Qt:=1$ is equivalent to $Dc:=1; Qt:=1$. The previous example is also written conveniently as:

$$\begin{align*}
\text{Actor.Exec} & \quad \Rightarrow \text{Actor.Init[i]} \\
\{ Rq := 0 \\ Pm,Dc := 1 \\ Qt := 0 \} & \quad \Rightarrow \{ Rq := 0 \\ Pm := 1 \\ Dc := Qt := 0 \} \\
\{ St := 1 \} & \quad \Rightarrow \{ St := 1 \} \\
\{ Ac := 0 \} & \quad \Rightarrow \{ Ac := 0 \} \\
\{ Rj := 1 \} & \quad \Rightarrow \{ Rj := 1 \} \\
\{ Sp := 0 \} & \quad \Rightarrow \{ Sp := 0 \}
\end{align*}$$

**Communication Acts and Facts**

For this aspect the $\Phi$ theory (section 1.2.3) is applicable, stating that the world is composed of Acta, Facta and Stata. Acta refers to any actions taken and Facta refers to the resulting facts. Actors communicate, are being presented an option (in the most elementary way it is a dialog box with option). Actors take a decision to choose one of the options, which is an act. The resulting fact is the choice being made public to other actors, which is a change of the communication state space.

Transaction patterns involve decisions being made by actors; an actor receives a request (Rq) and may reply with either a promise (Pm) or a decline (Dc). This principle is illustrated in the following example.

The decision ‘Promise | Decline’ is replied with a “Promise” denoted using function $Q()$:

$$Q(\text{Promise} \mid \text{Decline}) = \text{Promise} \quad \text{also noted as:} \quad Q(\text{Pm} | \text{Dc}) = \text{Pm}$$

This decision term indicates an asynchronous (meaning that there is no time imperative) but mandatory choice between the options promise and decline. The notation ‘a \mid b’ denotes an option of either a or b. The $Q()$ function can be called with respect to the one pfact to an actor as initiator and the multiple pfacts to actors as executors. The $Q()$ representation above does not include any index $i$ of pfacts, so it can be inferred that it is referring to the executor role. For the multiple initiator roles the following notation, including an index, is used:
Typically, in a production system environment there is more information passed to the actor by the function \( Q() \), such as transaction state and any information needed for a proper decision. The nature of this additional information is outside the scope of the transaction level.

### 5.5.2. Cfact Specifications of the Standard Transition Space

Cfacts are defined by the current state of the 8 elementary parameters, \((Rq, Pm, Dc, Qt, St, Rj, Ac, St)\). There is a limited set of allowed states; not all imaginable states are allowed. The set of 8 elementary cfacts allows for 256 different states. Analysis (section 4.5.10) delivers a set of 12 allowed states, \(S0\) to \(S11\), which are applicable to both single and aggregated transactions (section 4.3). The fact that there are 12 allowed states, a greater number than the 8 elementary parameters, implies that some states are permitted of which 2 or more parameters are set to true.

Each of the allowed states is unambiguous in its meaning. This implies that for any actor any previous states are irrelevant; they do not have to be 'remembered' or they do not have any influence anymore. The implementation of the transaction cancellation pattern (section 5.6) does not require additional parameters or states.

### 5.5.3. Request – Promise Patterns

State transitions are specified here, in line with cardinality law 3 (section 3.4). The numbering of axioms is random and hence is of no significance. These axioms are software programming specifications for the \( \Psi \) processor and subsequent verification (Appendix I).

[Axiom 1]

\[
\begin{align*}
\text{Actor.Init}(Rq, Pm, Dc, Qt & = 0) \\
\text{Actor.Init}(St, Ac, Rj, Sp & = 0) \\
\implies \text{Actort.Exec()} &\implies \\
\text{Actor.Q}(Rq & = 0) \\
\text{Actor.Init}(I) &\implies \\
\text{Actor.Init}(Rq & = 1) \\
\text{Actor.Init}(Pm : Dc : Qt : St : Ac : Rj : Sp : 0) &\implies \\
\end{align*}
\]

Axiom 1 states that if an actor is the root initiator, which is the case if this actor does not perform an executor role (i.e. when Exec() evaluates to False), they may issue the request.
to execute the root task. In this way Rq is set for initiator role number 0 (zero) as there is a sole root initiator and a sole root pfact. Initiator roles are zero-based indexed, indicated by the integer variable i. There is no option in the Q() function; the initiator has exclusively the option to issue a request at some convenient point in time, since the transaction executes asynchronously. At a more implementation-oriented level, an attribute states that an actor is self-initiating.

[Axiom 2]

\[
\begin{align*}
\text{Acto}r.\text{Exec} & : Rq = 0, \quad Pm = 1, \\
&Dc, Qt = 0, \\
&St, Ac, Rj, Sp = 0
\end{align*}
\]

\[\Rightarrow \forall i \in \text{actor}.
\begin{align*}
\text{Acto}r.\text{Init}[i] & : Rq = 0, \\
&\text{Pm} = 1, \\
&Dc = 0, \\
&Qt = 1, \\
&St, Ac, Rj, Sp = 0
\end{align*}
\]

In Axiom 2 it is stated that if an executor issues a promise, then for any initiator role where a quit and a promise have been issued, this quit should be overridden, reset, while the earlier set promise should be maintained.

Similarly follows:

[Axiom 3]

\[
\begin{align*}
\text{Acto}r.\text{Exec} & : Rq = 0, \quad Pm = 1, \\
&Dc, Qt = 0, \\
&St, Ac, Rj, Sp = 0
\end{align*}
\]

\[\Rightarrow \forall i \in \text{actor}.
\begin{align*}
\text{Acto}r.\text{Init}[i] & : Rq, Pm, Dc, Qt = 0, \\
&St, Ac, Rj, Sp = 0
\end{align*}
\]

Axiom 3 states that after, the executor’s promise, the request for all enabled subordinates is set if all parameters are reset.
5 Design of a Dynamic Logic Model for the \( \Psi \) Theory

[Axiom 4]

\[
\text{Actor.} \text{ Exec} \begin{cases} 
R_q = 0 \\
P_m = 1 \\
D_c, Q_t = 0 \\
S_t, A_c, R_j, S_p = 0
\end{cases} \ \Rightarrow \ \forall \ i \in \{ \text{Init} \} \begin{cases} 
R_q, P_m = 0 \\
D_c, Q_t = 1 \\
S_t, A_c, R_j, S_p = 0
\end{cases} \Rightarrow 
\begin{cases} 
R_q = 1 \\
D_c, Q_t = 0 \\
S_t, A_c, R_j, S_p = 0
\end{cases}
\]

Axiom 4 states that if an executor issues a promise, then for any initiator role where a decline has been accepted by a quit, both the decline and the quit will be reset while a (repeated) request is issued.

[Axiom 5]

\[
\text{Actor.} \text{ Exec} \begin{cases} 
R_q = 1 \\
P_m, D_c, Q_t = 0 \\
S_t, A_c, R_j, S_p = 0
\end{cases} \ \Rightarrow 
\begin{cases} 
R_q \Rightarrow 0 \\
P_m \Rightarrow 1 \\
D_c, Q_t = 0 \\
S_t, A_c, R_j, S_p = 0
\end{cases}
\]

Axiom 5 states that if an executor receives a request, a query will be made to decide either to promise or to decline, and one or the other attribute is set according to this decision.

[Axiom 6]

\[
\text{Actor.} \text{ Exec} \begin{cases} 
R_q = 1 \\
P_m, D_c, Q_t = 0 \\
S_t, A_c, R_j, S_p = 0
\end{cases} \ \Rightarrow 
\begin{cases} 
R_q \Rightarrow P_m \Rightarrow 0 \\
D_c \Rightarrow 1 \\
Q_t \Rightarrow S_t \Rightarrow 0 \\
A_c \Rightarrow R_j, S_p \Rightarrow 0
\end{cases}
\]

In axiom 6 the state transition is similar to axiom 5, except that the executor declines the request.
5 Design of a Dynamic Logic Model for the Ψ Theory

5.5.4. Request – Quit Patterns

Request / quit patterns occur after an executor issues a decline. The system issues the query ‘request | quit’ and the initiator decides to issue a quit. These transitions are also part of the negotiating phase, during which no task is executed and is defined by the attributes: State == 0; Accept == 0; Reject == 0; Stop == 0.

[Axiom 7]

\[
\forall_{i \in \mathbb{I}} \begin{cases}
Rq_i, Pm \equiv 0 \\
Dc \equiv 1 \\
Qt \equiv 0 \\
St_i, Ac_i, Rj_i, Sp \equiv 0
\end{cases} \Rightarrow \begin{cases}
\text{Actor}.Init[i](Rq_i \cup Qt) \equiv Rq \Rightarrow \\
\text{Actor}.Init[i](Rq \cup Qt) \equiv Qt \Rightarrow \\
\text{Actor}.Exec \equiv 0 \\
\text{Actor}.Exec \equiv 0
\end{cases}
\]

In axiom 7, a decline is issued while the promise as executor is still pending, then the initiator has to respond with a request or a quit query. If the reply is a request, the decline is overruled and the initial request is repeated.

If a decline is issued by (one of) their executors, the initiator has to respond to a query with either a repeated request or a quit. If a quit is returned then the actor has one role as executor and one or more roles as initiator to its subordinate executors.

[Axiom 8]

\[
\forall_{i \in \mathbb{I}} \begin{cases}
Rq, Pm \equiv 0 \\
Dc \equiv 1 \\
Qt \equiv 0 \\
St_i, Ac_i, Rj_i, Sp \equiv 0
\end{cases} \Rightarrow \begin{cases}
\text{Actor}.Init[i](Rq \cup Qt) \equiv Rq \Rightarrow \\
\text{Actor}.Exec \equiv 0 \\
\text{Actor}.Exec \equiv 0
\end{cases}
\]

Axiom 8 states that for one of the initiator roles, a decline has been issued, resulting in a query as to whether the request should be repeated or a quit should be issued. In case a quit
is issued, the executor issues a decline. This is the upward propagating process of a
termination or a cancellation of executable tasks.

From the executor the decline is passed upwards as in axiom 8, and for each of any other
initiators’ states (where i is not equal to j) axiom 9, axiom10 and axiom 11 apply.
Expressly, these axioms may be applicable simultaneously.

[Axiom 9]

\[
\forall_{i \neq j} \text{Actor.Init}[i] \Rightarrow
\begin{align*}
\text{Actor.Q}[j] & (\text{Rq} \mid \text{Qt}) = \text{Q} \Rightarrow \\
\text{Rq} & = 0 \\
\text{Dc} & = 1 \\
\text{Qt} & = 0 \\
\text{St}, \text{Ac}, \text{Rj}, \text{Sp} & = 0
\end{align*}
\]

In axiom 9 the decline from a subordinate actor results in a query to quit. For each initiator
role a quit is issued, in this case to the subordinate actor who issued a decline resulting in a
query to quit (where i equals j), but it is also applicable to any other actor (where i is not
equal to j) who has issued a decline.

Any other existing subordinate tasks, those for which no decline has been issued, can be in
several transaction states. Each of these states simply receives a quit from the initiator,
whilst maintaining its existing state. This allows for the quit to be withdrawn later after a
renewed request, hence restoring the previous state.
In axiom 10, for one of the initiator roles a decline results in a quit query. For any initiator role where a request has been issued earlier to which there was no response, this request is simply removed. In the period after a request is issued during which time the executor is considering a promise or a decline, if that request is then removed it is clear to the executor that they no longer have a promise / decline decision to make.

In axiom 11 it is stated that for one of the initiator roles a decline results in a quit query. For any initiator role where earlier a promise has been issued, the quit is set. In this case, no execution has yet taken place.
Axiom 12 states that for one of the initiator roles a decline resulted in a query of quit. For any initiator role where earlier a promise has been issued the quit is set.

Axiom 13 states that for one of the initiator roles a decline results in a quit query. For any initiator role where earlier a promise has been issued the quit is set. In this case, a state and a reject have been issued as part of the execution phase.
5.5.5. The Quit-Downward Rollback Propagation Mechanism

In the case that an executor issues a promise or a decline and his initiator decides to issue a quit in response, then the executor must acknowledge this quit before further downward propagation can take place. A 'quit' query is issued, to be followed by an acknowledgement from the executor. This differs from the decline-quit pattern in that, in this case, the quit originates from a parent (or grandparent) initiator. This is a rollback mechanism, one of the two rollback mechanisms, in which the earlier negotiation results are reverted and the rollback propagates until the terminal transactions (at the end of the tree), or those transactions where no negotiation has taken place, are reached. The transitions state = 0; accept = 0; reject = 0; stop = 0 are all part of the negotiating phase.

[Axiom 14]

\[
\begin{align*}
\text{Actor}.\text{Exec} & \quad \begin{cases} \text{Rq}, \text{Pm} = 0 \\
\text{Dc}, \text{Qt} = 1 \\
\text{St}, \text{Ac}, \text{Rj}, \text{Sp} = 0
\end{cases} \\
\Rightarrow & \quad \begin{cases} \text{Actor}.\text{Q}(\text{Qt}) = \text{Qt} \Rightarrow \\
\text{Actor}.\text{Exec} & \quad \begin{cases} \text{Rq} := \text{Pm} := \text{Dc} := \text{Qt} := 0 \\
\text{St} := \text{Ac} := \text{Rj} := \text{Sp} := 0
\end{cases}
\end{cases}
\end{align*}
\]

In axiom 14, it is stated that for an executor who issues a decline followed by a quit from his initiator, there is an acknowledgement needed – no options to choose between, just an acknowledge – from the actor resulting in all parameters reset, i.e. a return to the default state. In this case no prior execution has taken place.

[Axiom 15]

\[
\begin{align*}
\text{Actor}.\text{Exec} & \quad \begin{cases} \text{Rq}, \text{Pm} = 0 \\
\text{Dc}, \text{Qt} = 1 \\
\text{St}, \text{Ac}, \text{Rj}, \text{Sp} = 0
\end{cases} \\
\Rightarrow & \quad \begin{cases} \text{Actor}.\text{Q}(\text{Qt}) = \text{Qt} \Rightarrow \\
\forall i \in \{\text{Actor}.\text{Init}[i] & \quad \begin{cases} \text{Rq} = 1 \\
\text{Pm}, \text{Dc}, \text{Qt} = 0 \\
\text{St}, \text{Ac}, \text{Rj}, \text{Sp} = 0
\end{cases} \\
\Rightarrow & \quad \begin{cases} \text{Actor}.\text{Init}[i] & \quad \begin{cases} \text{Rq} := \text{Pm} := \text{Dc} := \text{Qt} := 0 \\
\text{St} := \text{Ac} := \text{Rj} := \text{Sp} := 0
\end{cases}
\end{cases}
\end{cases}
\end{align*}
\]

In axiom 15 a quit – acknowledge results, for each initiator role where a request has been issued, in a withdrawal – a reset – of the request. The subordinate executor may or may not have been aware of this pending request but in any case did not respond. The fact that the pending request disappeared does not cause any ambiguity and implies that the subordinate executor will not take any action.
5 Design of a Dynamic Logic Model for the $\Psi$ Theory

[Axiom 16]

$$\forall i \in \text{Actor} \cdot \text{Exec} \left( \begin{array}{l}
Rq, Pm = 0 \\
Dc, Qt = 1 \\
St, Ac, Rj, Sp = 0
\end{array} \right) \Rightarrow \\
\text{Actor} \cdot \text{Q}(Qt) = Qt \Rightarrow \\
\left[ \begin{array}{l}
\text{Actor} \cdot \text{Init}[i] \\
\Rightarrow \text{Actor} \cdot \text{Init}[i]
\end{array} \right]$$

In axiom 16 a quit – acknowledge results in an additional quit being set, for each initiator role where a decline has been issued. In this case, the initiator receives a quit. Note that the initiator is not aware that in this case the quit results from a downward propagation and not in response to his initial decline.

[Axiom 17]

$$\forall i \in \text{Actor} \cdot \text{Exec} \left( \begin{array}{l}
Rq, Pm = 0 \\
Dc, Qt = 1 \\
St, Ac, Rj, Sp = 0
\end{array} \right) \Rightarrow \\
\text{Actor} \cdot \text{Q}(Qt) = Qt \Rightarrow \\
\left[ \begin{array}{l}
\text{Actor} \cdot \text{Init}[i] \\
\Rightarrow \text{Actor} \cdot \text{Init}[i]
\end{array} \right]$$

In axiom 17 a quit – acknowledge as executor results, for each initiator role where a promise has been issued, in an additional quit being set. In this case no intermediate execution has taken place.
[Axiom 18]

In axiom 18 a quit – acknowledge as executor results, for each initiator role where a promise has been issued, in an additional quit being set. In this case, a state has been issued which is kept.

[Axiom 19]

In axiom 19 a quit – acknowledge results, for each initiator role where a promise has been issued, in an additional quit being set.
In a similar fashion to axioms 15 to 19, the same transitions must be specified for the condition in which a quit is issued while a promise is issued. This is specified in axiom 20 and in axioms 34 to 39.

5.5.6. Execution Phase Transition Patterns

As yet, the aggregated request-promise and decline-quit patterns have been investigated; these are relatively simple patterns to describe, because no actual execution of a task has yet taken place; they comprise only communication and negotiation. Introducing the element of execution presents the question as to how the collision between a downward directed propagated quit should be handled once the bottom-up execution of any subordinate task(s) is actually underway. In the real world, this is an important issue. Even if a task has not been accepted, there are intermediate results. The handling of these intermediate results may require special tasks, which have to be designed at modeling time.

The execution phase in an aggregated system is far more complex than the negotiation phase. Execution can take place only when all subordinate tasks have been executed and the results have been accepted. This is represented by an additional column (‘each initiator if’) that states that for each subordinate task (with the role of initiator) the accept parameter is set. The condition that for each initiator role the tasks have been completed and accepted cannot change at this point so for any subsequent state transitions this condition can be omitted.

The query is here a (request for) an acknowledgement. When the task has been completed then the acknowledgement is issued by the executor; if it is not yet completed then the query waits. When an accept is set, it signifies that the execution of that task has been completed, the transaction pattern has also completed and will not be subject to any changes; the production fact exists.
Axiom 21 states that if an executor issues a promise and all existing enabled subordinate tasks of that executor have been completed and accepted, then a query is issued to the executor for him to acknowledge whether his own task is ready to be stated. Once the executor acknowledges this, the state is issued by the actor, i.e. the parameter state is set; \( St := 1 \).

The acknowledgement of a quit by an executor \((Actor.Q(Qt) == Qt)\) during the execution phase always results in a reset of all parameters to their default state. However, it is mandatory to specify and handle each of the state transitions of the execution phase separately. This allows for separate event code modules, to enable the executor to handle the results of his execution for each state.
5 Design of a Dynamic Logic Model for the $\Psi$ Theory

[Axiom 22]
\[
\begin{align*}
\text{Actor}.\text{Exec} & \equiv \\
Rq & = 0 \\
Pm & = 1 \\
Dc & = 0 \\
Qt, St & = 1 \\
Ac, Rj, Sp & = o \\
\Rightarrow & \\
\text{Actor}.\text{Q}(Qt) & = Qt \Rightarrow \\
\text{Actor}.\text{Exec} & \equiv \\
Rq & = Pm = Dc = Qt = o \\
St & = Ac = Rj = Sp = o
\end{align*}
\]

In axiom 22 it is stated that if an executor receives a quit after a state has been issued, then after an acknowledgement of the quit all parameters are reset.

[Axiom 23]
\[
\begin{align*}
\text{Actor}.\text{Exec} & \equiv \\
Rq & = 0 \\
Pm & = 1 \\
Dc & = 0 \\
Qt, St & = 1 \\
Ac & = 0 \\
Rj & = 1 \\
Sp & = o \\
\Rightarrow & \\
\text{Actor}.\text{Q}(Qt) & = Qt \Rightarrow \\
\text{Actor}.\text{Exec} & \equiv \\
Rq & = Pm = Dc = Qt = o \\
St & = Ac = Rj = Sp = o
\end{align*}
\]

In axiom 23, it is stated that if an executor receives a quit, after the issuing of a state followed by reject, then after acknowledgement of the quit all parameters are reset.

The situation in which a quit is issued, after the issuing of an accept, does not exist (the actual performance has been created, accepted and exists as a fact). After an accept from the initiator, the executor immediately resets all negotiating parameters (i.e. $Rq = o; Pm = o; Dc = o; Qt = o$) and the initiator will not issue a quit. The question of what should be done with the completed performance is not handled at this level.
In axiom 24 it is stated that if an initiator accepts a state - the task has been performed and accepted - all negotiation parameters are reset except the accept parameter, $Ac = 1$.

5.5.7. **State Transitions for a Reject after a State**

After a state of an executor, the initiator has two options, either to reject or to accept. After a reject the following patterns may occur.

In axiom 24 it is stated that if an initiator accepts a state - the task has been performed and accepted - all negotiation parameters are reset except the accept parameter, $Ac = 1$. 

[Axiom 25]
Axiom 25 states that a reject after a state results in reject being set while the state being set is kept.

\[
\begin{align*}
\text{Actor.Exec} & \quad \begin{cases}
Rq = 0 \\
Pm = 1 \\
Dc, Qt = 0 \\
St = 1 \\
Ac = 0 \\
Rj = 1 \\
Sp = 0
\end{cases} \\
\Rightarrow & \quad \begin{cases}
\text{Actor.Q} (St | Sp) = St \Rightarrow \\
Rq := 0 \\
Pm := 1 \\
Dc := Qt := 0 \\
St := 1 \\
Ac := Rj := Sp := 0
\end{cases}
\end{align*}
\]

Axiom 26 states that an executor, after having received a reject, may decide to state (again) or to stop, optionally after some modification of the task executed; here the executor decides to keep the state==1 and to reset the reject.

\[
\begin{align*}
\text{Actor.Exec} & \quad \begin{cases}
Rq = 0 \\
Pm = 1 \\
Dc, Qt = 0 \\
St = 1 \\
Ac = 0 \\
Rj = 1 \\
Sp = 0
\end{cases} \\
\Rightarrow & \quad \begin{cases}
\text{Actor.Q} (St | Sp) = Sp \Rightarrow \\
Rq := Pm := Dc := Qt := 0 \\
St := 0 \\
Ac := Rj := Sp := 0
\end{cases}
\end{align*}
\]

Axiom 27 states that an executor, after having received a reject, may decide to stop; in this case the executor decides to stop. The parameter stop is set and the promise is removed.

5.5.8. The Stop Upward Rollback Propagation Mechanism

After a state command is followed by a reject by the executor a stop command is issued, meaning "I cannot deliver the requested result". At the same time the initial promise is also withdrawn. The initiator must also acknowledge this explicitly for this transaction \((\text{actor.Q[i]}(Sp) = Sp)\) before the transaction continues. The initiator who acknowledges a stop instruction reverses his role to executor, issuing a decline and resetting the earlier
issued promise to false, to his initiator. The rollback process propagates upward as described earlier. Unlike the previously described downward rollback - which cannot be reverted and propagates until the terminal transaction or a transaction that has not yet started negotiating – this upward rollback can be reverted at higher transaction levels by the act of issuing a new request. This complies with common understanding, as at a higher level the scope of the actors is greater than at lower levels and in this way a repeated request may make sense.

In axioms 28 to 33, a stop issued by one of the executors is handled by the initiator. The consequences of a stop apply to the executor who issues the stop, as well as to the initiator who receives the stop, both in his role as executor and in his roles as initiator for the other executors.

**[Axiom 28]**

\[
\forall i \left[ \begin{array}{c}
\text{Actor.Init}(i) \left[ \begin{array}{c}
Rq, Pm, Dc, Qt = 0 \\
St, Ac, Rj = 0 \\
Sp = 1
\end{array} \right] \Rightarrow \\
\text{Actor.Init}(i) \left[ \begin{array}{c}
Rq = 0, Pm = 0, Dc = 0, Qt = 0 \\
St = 0, Ac = 0, Rj = 0, Sp = 0
\end{array} \right]
\end{array} \right]
\]

Axiom 28 states that an initiator, after having received a stop, acknowledges this and, in this specific initiator role, resets the stop and restores the default state. The negotiation process can be repeated as specified in axiom 3.

**Implementation issue:**

The state change of axiom 28 must be carried out in software implementation after the state changes as specified in axiom 29 to axiom 32, otherwise the calculations of the conditions for these axioms to be executed may be missed. Furthermore, the acknowledgement of the stop should be executed only once. The correct practical solution in software is an implementation in a single conditional block, testing first for \( Rq, Pm, Dc, Qt = 0; St, Ac, Rj = 0; Sp = 1 \) and then modifying the state changes for each axiom and each role.
Axiom 29 states that, under the same conditions as axiom 28, an initiator acknowledges a stop issued from one of his subordinate executors, upon which he changes role as executor and issues an upward decline command. The communication between the initiator[i] in his executor role is still in the negotiation phase where only a promise has been issued. This is not a violation of the transaction axiom. The executor issues first a promise but then replaces this promise by a decline, which is allowed since the state of the transaction did not change and the initiator did not act upon this promise instruction.

If in the executor role a quit, request or decline command is pending then the upward decline propagation does not apply. Note that axiom 29 is similar to axiom 8. The acknowledgement of a stop instruction by an initiator bears consequences for any other sibling initiator roles, if any. The other initiator roles receive a quit command as specified in axioms 30 to 32. This instance of quit starts the downward quit propagation mechanism implemented by axioms 14 to 20. It should be noted here that the other parameters of the execution phase are not modified; the executor is aware of both the quit instruction and the previous state, which eliminates any ambiguity.
Axiom 30 refers to the situation where one of the initiator roles issues a stop command. Once this stop has been acknowledged, a quit is also issued if for any other initiator role only a promise has been issued. This is similar to axiom 11.

Axiom 31 calculates if for one of the initiator roles a stop instruction has been issued. After the acknowledgement of that stop, if a promise and a state have been issued for any other initiator role, a quit is additionally issued. Axiom 31 is similar to axiom 12.
Axiom 32 deals with the situation where, for one of the initiator roles, a stop has been issued. After the acknowledgement of the stop command, if for any other initiator role a promise, state and a reject have been issued, a quit is additional issued. This is similar to axiom 13.

Axiom 33 states that when a stop has been issued for any initiator role, after the acknowledgement of this stop, any initiator-issued request is simply withdrawn. This is similar to axiom 10.

In axiom 20, the executor has declared a promise but then receives a quit instruction, whereupon he acknowledges this quit and resets all parameters to the initial state. Note that
axioms 14 to 19 share the same initial condition followed by a quit acknowledgement; furthermore, they are executed simultaneously, but deal with different initiator states. These are treated specifically in axioms 34 to 38.

[Axiom 34]

\[
\begin{align*}
\text{Actor.Exec} & : \left\{ \begin{array}{l}
Rq = 0 \\
Pm = 1 \\
Dc = 0 \\
Qt = 1 \\
St, Ac, Rj, Sp = 0
\end{array} \right. \\
\Rightarrow \\
\forall i \in I : \left[ \begin{array}{l}
\text{Actor.Qt} = Qt \\
\Rightarrow \\
\text{Actor.Init}[i] \left\{ \begin{array}{l}
Rq = 1 \\
Pm, Dc, Qt = 0 \\
St, Ac, Rj, Sp = 0
\end{array} \right.
\end{array} \right]
\end{align*}
\]

In axiom 34, a quit – acknowledge pattern results for each initiator role where a request has been issued in a withdrawal – a reset – of the request. The subordinate executor may or may not have been aware of this pending request but has not yet responded. The fact that the pending request disappeared does not cause any ambiguity and implies that the subordinate executor will not take any action. Note that in subordinate transactions, no request has been issued such that these cfact states are still default Cfact S0. This axiom is similar to axiom 15.

[Axiom 35]

\[
\begin{align*}
\text{Actor.Exec} & : \left\{ \begin{array}{l}
Rq = 0 \\
Pm = 1 \\
Dc = 0 \\
Qt = 1 \\
St, Ac, Rj, Sp = 0
\end{array} \right. \\
\Rightarrow \\
\forall i \in I : \left[ \begin{array}{l}
\text{Actor.Qt} = Qt \\
\Rightarrow \\
\text{Actor.Init}[i] \left\{ \begin{array}{l}
Rq, Pm = 0 \\
Dc = 1 \\
Qt = 0 \\
St, Ac, Rj, Sp = 0
\end{array} \right.
\end{array} \right]
\end{align*}
\]

In axiom 35 a quit – acknowledge sequence results for each initiator role where a decline has been issued. In this case, the initiator receives a quit command. Note that the initiator is not aware that in this case the quit is resulting from a downward propagation and not in response to his initial decline. This axiom is similar to axiom 16.
In axiom 36, a quit–acknowledge as executor results in an additional quit being set for each initiator role where a promise instruction has been issued. This is similar to axiom 17.

In axiom 37 a quit–acknowledge as executor results in an additional quit command being set for each initiator role where a promise has been issued. In this axiom, a state command has been issued, which is kept. This is similar to axiom 18.
In axiom 38, a quit – acknowledge results, for each initiator role where a promise has been issued, in an additional quit being set. This is similar to axiom 19.

Note:
An initiator cannot issue a quit to a transaction where an accept has been issued (cfact S11, section 5.5.9). This transaction has completed with some available production; and accepted by both initiator and executor. It is a fact according to the \( \Phi \) theory (section 1.2.3), that cannot be "destroyed by denying its existence".

5.5.9. Specification of the Cfact State Space

Investigation of the set of allowed cfact states from the state transitions of the previous section results in a set of 12 cfact states \{S0, …, S11\}, for any cfact in an aggregated model; other cfact states are not allowed. This is precisely in line with the second cardinality law (section 2.4.7). Each of these cfact states has an unambiguous position in software\(^{86}\) for both the initiator and the executor. This set of states can also be derived by analysis of the state transactions of the elementary DEMO model. This is important as any actor acts exclusively on the state of this transaction; the states of parent or child-transactions are irrelevant. The transaction axiom is valid for aggregated systems and the elementary transaction. This implies that the number of cfact states and their meanings for aggregated systems is equal and identical to the number of cfact states and meanings for the elementary transaction.

Business rules, or action model rules, as specified in section 6.3.3 are to be executed when a specific state is reached. The following exemplifies the notation that is used to specify an action model rule that is to be executed at the defined state transition:

\(^{86}\) We use the DEMO symbols also for corresponding software objects.
Action Rule: \( \text{OnDefault} \{ \text{Line} \} \text{No} \)

The symbol \( \text{OnDefault} \) denotes the beginning of the action model rule. On the \( \text{OnDefault} \) event the set \((0,1..n)\) of \( \{ \text{Line} \} \) with each an action model rule is executed. These \( \text{Line} \) elements are defined by the action model processor business rules and instruction set as specified in section 6.3.3. The \( \text{No} \) symbol denotes the end of the action model section.

Set of allowed states with the related action model rule

**Cfact S0**

\[
\begin{align*}
Rq, Pm, Dc, Qt &\Rightarrow 0 \\
St, Ac, Rj, Sp &\Rightarrow 0
\end{align*}
\]

\( [\text{Cfact S0}] \)

Cfact state S0 is the default state, before the transaction execution starts.

**Action Rule S0**: \( \text{OnDefault} \{ \text{Line} \} \text{No} \)

Since a transaction may reach its default state after previous state changes, an action model rule event must be defined.

**Cfact S1**

\[
\begin{align*}
Rq &\Rightarrow 1 \\
Pm, Dc, Qt &\Rightarrow 0 \\
St, Ac, Rj, Sp &\Rightarrow 0
\end{align*}
\]

\( [\text{Cfact S1}] \)

Cfact state S1 states that a request instruction is issued by the initiator.

**Action Rule S1**: \( \text{OnRequested} \{ \text{Line} \} \text{No} \)
Cfact S2
\[\left\{\begin{array}{l}
Rq = 0 \\
Pm = 0 \\
Dc = 1 \\
Qt = 0 \\
St, Ac, Rj, Sp = 0
\end{array}\right.\]  
[Cfact S2]

Cfact state S2 states that a decline command is issued by the executor.

**Action Rule S2:**  OnDeclined { Line } No

Cfact S3
\[\left\{\begin{array}{l}
Rq = 0 \\
Pm = 1 \\
Dc, Qt = 0 \\
St, Ac, Rj, Sp = 0
\end{array}\right.\]  
[Cfact S3]

Cfact state S3 states that a promise is issued by the executor.

**Action Rule S3:**  OnPromised { Line } No

Cfact S4
\[\left\{\begin{array}{l}
Rq = 0 \\
Pm = 1 \\
Dc = 0 \\
Qt = 1 \\
St, Ac, Rj, Sp = 0
\end{array}\right.\]  
[Cfact S4]

Cfact state S4 states that the initiator issued a quit command during a pending promise.

**Action Rule S4:**  OnQuitPromised { Line } No
Cfact S5
\[
\begin{align*}
Rq, Prm &= 0 \\
Dc, Qt &= 1 \\
St, Ac, Rj, Sp &= 0
\end{align*}
\]  
[Cfact S5]

Cfact state S5 states that the initiator issued a quit command, with a pending decline.

**Action Rule S5:** OnQuitDeclined \{ Line \} No

Cfact S6
\[
\begin{align*}
Rq &= 0 \\
Prm &= 1 \\
Dc, Qt &= 0 \\
St &= 1 \\
Ac, Rj, Sp &= 0
\end{align*}
\]  
[Cfact S6]

Cfact state S6 states that a state command is issued by the executor.

**Action Rule S6:** OnStated \{ Line \} No

Cfact S7
\[
\begin{align*}
Rq &= 0 \\
Prm &= 1 \\
Dc, Qt &= 0 \\
St &= 1 \\
Ac &= 0 \\
Rj &= 1 \\
Sp &= 0
\end{align*}
\]  
[Cfact S7]

Cfact state S7 states that the initiator issued a reject, after a state issued by the executor.

**Action Rule S7:** OnRejected \{ Line \} No
Cfact S8

\[
\begin{align*}
Rq &= 0 \\
\text{Pm} &= 1 \\
Dc &= 0 \\
Qt &= 1 \\
St &= 1 \\
\{Ac, Rj, Sp\} &= 0
\end{align*}
\]

[Cfact S8]

Cfact state S8 states that the initiator issued a quit, after a state issued by the executor.

**Action Rule S8:** OnQuitStated [Line] No

Cfact S9

\[
\begin{align*}
Rq, \text{Pm}, Dc, Qt &= 0 \\
St, Ac, Rj &= 0 \\
Sp &= 1
\end{align*}
\]

[Cfact S9]

Cfact state S9 states that the executor issued a stop, after a reject command from the initiator.

**Action Rule S9:** OnStopped [Line] No
Cfact S10

\[
\begin{align*}
Rq &= 0 \\
Pm &= 1 \\
Dc &= 0 \\
Qt &= 1 \\
St &= 1 \\
Ac &= 0 \\
Rj &= 1 \\
Sp &= 0
\end{align*}
\]

[Cfact S10]

Cfact state S10 states that the initiator issued a quit command, after the initiator had issued a reject command.

**Action Rule S10: OnQuitRejected [ Line ] No**

Cfact S11

\[
\begin{align*}
Rq, Pm, Dc, Qt &= 0 \\
St &= 0 \\
Ac &= 1 \\
Rj, Sp &= 0
\end{align*}
\]

[Cfact S11]

Cfact state S11 states that the initiator issued an accept and that the transaction is completed.

**Action Rule S11: OnAccepted [ Line ] No**

There are 4 OnQuitXXX states, depending on the state present when the quit is issued. There are 7 remaining states (OnRequested, OnPromised, etc.) with the default state S0. In total, there are 12 states.
5.5.10. Graphical Representation of Cfact State and Transition Spaces

Figure 5.9 Cfact State and Transition spaces.
A graphical representation of all allowed cfact states – S0..S11 – and standard state transitions with their indices is shown in figure 5.9. Note that this cannot be represented by a fixed 256 X 256 matrix. First, the dimension of the matrix varies, depending on the number of transactions and at each matrix element a calculation has to be made for the Q() function.

The scheme in figure 5.9 shows no visual anomalies, such as abandoned states and deadlocks. Soundness is a quality that implies that for any state that can be reached, any other state can be reached. Each state, except S11, can be reached by one or more state transitions and can be left by other state transitions. State S0 is the initial state. State S11 is terminal (Ac==1) meaning that the pfact has been executed and accepted, it is a fact, and this state can no longer be abandoned. Figure 5.9 shows soundness.

The cfact states shown in figure 5.9 do not necessarily belong to the same transaction. An 'arrow' from an axiom means that that axiom assigns to this state. An arrow to an axiom means that that cfact state is used in the calculation in the conditional part of that axiom. The state transition axioms cannot be represented by an 8 X 8 matrix. This figure is therefore not the ultimate result of the state transition axioms, its purpose is to show easily that there is soundness, except for S0 and S11.

5.6. Transaction Cancellation Patterns

The transaction axiom also provides a transaction pattern for cancellation of all earlier decisions taken in the standard transaction pattern. It is an extension, an addition to the standard transaction pattern. This pattern is empirically supported by real world observations that actors may decide, at any time and for any reason known to them, to cancel the last basic coordination act. This cancel does not imply that any subsequent transaction communication has been terminated; any communication will proceed as defined by the axioms.

In this pragmatic implementation – fit for purpose - is chosen for a slightly different implementation. A cancel by an actor is implemented as the rollback of the default process for all transactions by issuing a decline or a quit. It is not the withdrawal of the last communication act for a specific role. Identification of the “best” implementation of cancellation patterns is subject to empirical assessment of practical applications. Specific desired implementations can easily be implemented as a business rule: OnQuit [Line] No . The expressiveness of the chosen implementation is assumed to be adequate, any desired pattern can be implemented.

In these extensions and implementations of the standard transaction pattern, a cancellation can be specified directly without any additional cancellation parameters. If an initiator issues a quit to all child actors and a decline as executor then a de facto cancellation is
implemented. The effect of the cancellation for an actor is identical to receiving a quit as executor from the parent actor. This implies first that only for that specific actor is it visible that a cancel has been issued, while at all other levels the actors cannot distinguish a cancel from a rollback propagation (sections 4.5.5, 4.5.8). Second, it implies that there is no need for additional cancel parameters (such as allow) in addition to the standard eight transaction parameters.

Each cancel axiom specifies one cfact state transition. During any Cancel() operation, several axioms simultaneously apply to the related cfacts of that actor. The first group of cancellation pattern is for the 12 transaction states as executor. Some cancellations do not result in a state change because this is unnecessary, such as the case in which the transaction state was already in a corresponding state. This implies that in general, for any implementation, the execution of state transition axioms should be atomic in time. It is clear that while a cancellation may or may not change the transaction state, a second cancellation action clearly will never result in any state change.

There is a fundamental difference between actors decisions such as 'quit or decline', 'accept or reject' and the transaction cancellation. A cancel can be issued at any random moment in time by an actor, without using the Q() function. The Q() function is a member function of an actor and is driven by the $\Psi$ processor. Cancel is not driven by the $\Psi$ processor, but is driven by the actor by calling a member function actor.Cancel() of the actor entity at any random point in time. Depending on the transaction state S0..S11 at that time, the transaction continues.

### 5.6.1. Executor Cancellation Patterns

The executor cancellation patterns are similar to other state transitions. Its effect here is to induce the upward rollback patterns and downward quit patterns. For each allowed state (S0..S11) a transition is specified. The following notation is in line with previously defined convention, showing an object O with member function F and parameter set p implemented by a set of statements:

$$\begin{align*}
O.F(p): & \text{..} \\
& \text{Statement}
\end{align*}$$

For cfact state S0:
5 Design of a Dynamic Logic Model for the $\Psi$ Theory

\[
\begin{align*}
(R_q, P_m, D_c, Q_t &\equiv o) \\
S_t, A_c, R_j, S_p &\equiv o
\end{align*}
\]  

[Cfact S0]

which is the default state, a cancel instruction, \texttt{Cancel()}, for an executor has no meaning or effect.

[Axiom 39]

\texttt{Cancel()} of executor cfact state S1

\[
\begin{align*}
\text{Actor} . \text{Cancel}() : \text{Actor} . \text{Exec} \\
R_q \equiv 1 &
\end{align*}
\]

\[
\begin{align*}
P_m, D_c, Q_t &\equiv o \\
S_t, A_c, R_j, S_p &\equiv o
\end{align*}
\]

\[
\begin{align*}
\Rightarrow \text{Actor} . \text{Exec} \\
R_q &\equiv P_m \equiv o \\
D_c &\equiv 1 \\
Q_t &\equiv o \\
S_t &\equiv A_c \equiv R_j \equiv S_p \equiv o
\end{align*}
\]

\texttt{Cancel()} of executor cfact state S1, a request has been issued. After a \texttt{Cancel()}, a decline is issued by the executor, as given in cfact S2.

\texttt{Cancel()} of executor cfact state S2:

\[
\begin{align*}
(R_q, P_m &\equiv o) \\
D_c &\equiv 1 \\
Q_t &\equiv o \\
S_t, A_c, R_j, S_p &\equiv o
\end{align*}
\]

[Cfact S2]

Here the executor has issued a decline. A \texttt{Cancel()} is ignored.

[Axiom 40]

\texttt{Cancel()} of executor cfact state S3:

\[
\begin{align*}
\text{Actor} . \text{Cancel}() : \text{Actor} . \text{Exec} \\
R_q &\equiv o \\
P_m &\equiv 1 \\
D_c, Q_t &\equiv o \\
S_t, A_c, R_j, S_p &\equiv o
\end{align*}
\]

\[
\begin{align*}
\Rightarrow \text{Actor} . \text{Exec} \\
R_q &\equiv P_m \equiv o \\
D_c &\equiv 1 \\
Q_t &\equiv o \\
S_t &\equiv A_c \equiv R_j \equiv S_p \equiv o
\end{align*}
\]

\texttt{Cancel()} of executor cfact state S3, the executor has issued a promise, after which the promise is withdrawn and replaced by a decline which results in cfact state S2.
[Axiom 41]
Cancel() of executor cfact state S4:

\[
\begin{align*}
\text{Actor.Cancel( \_ )} & \cdot \text{Actor.Exec} \\
\{ \begin{array}{l}
Rq := 0 \\
Pm := 1 \\
Dc := 0 \\
Qt := 1 \\
St, Ac, Rj, Sp := 0
\end{array} & \Rightarrow \begin{array}{l}
Rq := Pm := 0 \\
Dc := Qt := 0 \\
St := Ac := 0 \\
Rj := Sp := 0
\end{array}
\end{align*}
\]

Cancel() of executor cfact state S4, the initiator has issued a quit command during a pending promise. The executor returns to the default state S0.

[Axiom 42]
Cancel() of executor cfact state S5:

\[
\begin{align*}
\text{Actor.Cancel( \_ )} & \cdot \text{Actor.Exec} \\
\{ \begin{array}{l}
Rq := Pm := 0 \\
Dc := Qt := 1 \\
St, Ac, Rj, Sp := 0
\end{array} & \Rightarrow \begin{array}{l}
Rq := Pm := 0 \\
Dc := Qt := 0 \\
St := Ac := 0 \\
Rj := Sp := 0
\end{array}
\end{align*}
\]

Cancel() of executor cfact state S5, the initiator issued a quit command after a pending decline. The executor returns to the default state S0.

[Axiom 43]
Cancel() of executor cfact state S6:

\[
\begin{align*}
\text{Actor.Cancel( \_ )} & \cdot \text{Actor.Exec} \\
\{ \begin{array}{l}
Rq := 0 \\
Pm := 1 \\
Dc := Qt := 0 \\
St := 1 \\
Ac, Rj, Sp := 0
\end{array} & \Rightarrow \begin{array}{l}
Rq := Pm := 0 \\
Dc := Qt := 0 \\
St := Ac := Rj := 0 \\
Sp := 1
\end{array}
\end{align*}
\]

Cancel() of executor cfact state S6, the executor first issued a state instruction, then withdraws the state and issues a stop command, as in cfact state S9.
5 Design of a Dynamic Logic Model for the $\Psi$ Theory

[Axiom 44]
Cancel() of executor cfact state S7:

$$\begin{align*}
\text{Actor}.\text{Cancel}(\cdot) &\rightarrow \text{Actor}.\text{Exec} \\
\text{Actor}.\text{Exec} &\Rightarrow \text{Actor}.\text{Exec}
\end{align*}$$

$$\begin{align*}
Rq &= 0 \\
Pm &= 1 \\
Dc &= Qt = 0 \\
St &= 1 \\
Ac &= 0 \\
Rj &= 1 \\
Sp &= 0
\end{align*}$$

$$\begin{align*}
Rq &\Rightarrow Pm := 0 \\
Dc &\Rightarrow Qt := 0 \\
St &\Rightarrow Ac := Rj := 0 \\
Sp &:= 1
\end{align*}$$

Cancel() of executor cfact state S7, the initiator issued a reject command after a state from the executor. The executor issues a stop, as in cfact state S9.

[Axiom 45]
Cancel() of executor cfact state S8:

$$\begin{align*}
\text{Actor}.\text{Cancel}(\cdot) &\rightarrow \text{Actor}.\text{Exec} \\
\text{Actor}.\text{Exec} &\Rightarrow \text{Actor}.\text{Exec}
\end{align*}$$

$$\begin{align*}
Rq &= 0 \\
Pm &= 1 \\
Dc &= 0 \\
Qt &= 1 \\
St &= 1 \\
Ac &= Rj = Sp = 0
\end{align*}$$

$$\begin{align*}
Rq &\Rightarrow Pm := 0 \\
Dc &\Rightarrow Qt := 0 \\
St &\Rightarrow Ac := 0 \\
Rj &:= Sp := 0
\end{align*}$$

Cancel() of executor cfact state S8, the initiator issued a quit command after a state from the executor. The executor sets the default state S0.

Cancel() of executor cfact state S9:

$$\begin{align*}
Rq, Pm, Dc, Qt &= 0 \\
St, Ac, Rj &= 0 \\
Sp &= 1
\end{align*}$$

[Cfact S9]
The executor has issued a stop command; the cancel instruction is ignored.
5 Design of a Dynamic Logic Model for the Ψ Theory

[Axiom 46]

Cancel() of executor cfact state S10:

\[
\begin{align*}
\text{Actor.} & \text{Cancel} \left( \{ \text{Actor.} \text{Exec} \text{.} \text{Cancel} \} \right) \Rightarrow \text{Actor.} \text{Exec} \\
& \left\{ \begin{array}{l}
Rq = 0 \\
Pm = 1 \\
Dc = 0 \\
Qt = 1 \\
St = 1 \\
Ac = 0 \\
Rj = 1 \\
Sp = 0
\end{array} \right. \\
& \left\{ \begin{array}{l}
Rq := Pm := 0 \\
Dc := Qt := 0 \\
St := Ac := 0 \\
Rj := Sp := 0
\end{array} \right.
\end{align*}
\]

Cancel() of executor cfact state S10, the initiator issued a quit following a state and reject; in this way the executor sets the default state S0.

Cancel() of executor cfact state S11:

\[
\begin{align*}
\left\{ \begin{array}{l}
Rq, Pm, Dc, Qt = 0 \\
St = 0 \\
Ac = 1 \\
Rj, Sp = 0
\end{array} \right. \\
& \Rightarrow \text{Cfact S11}
\end{align*}
\]

The initiator issued an accept and thus the transaction is completed. A completed transaction in reality is a fact that cannot be destroyed, so the transaction remains completed and a cancel at this stage is ignored, as in cfact state S11.

For each of the 12 executor states a Cancel() induced state transition has been specified. Each of these Cancel() state changes induce the upward rollback mechanism if there was not already an upward rollback cycle in progress.

5.6.2. Initiator Cancellation Patterns

A cancellation for an initiator role induces a downward rollback cycle or a withdrawal of an earlier request. This is similar to the induction of rollback cycles through a quit or stop command in the standard patterns, but in this case induced by the Cancel() call. All child-transactions must rollback recursively, there is no reversal possible.
5 Design of a Dynamic Logic Model for the $\Psi$ Theory

**Cancel()** of initiator cfact state S0:
\[
\begin{aligned}
(Rq, Pm, Dc, Qt & \equiv 0) \\
(St, Ac, Rj, Sp & \equiv 0)
\end{aligned}
\]

[Cfact S0]

which is the default state, a cancel has no meaning, resulting in cfact state S0.

[Axiom 47]
**Cancel()** of initiator cfact state S1:

\[
\text{Actor.Cancel}(i) : \forall i \in I_i
\begin{aligned}
& \text{Actor.Init}[i] \\
& \Rightarrow \text{Actor.Init}[i]
\end{aligned}
\]

For cfact state S1, a request instruction has been issued and after a **Cancel()** the initiator withdraws the request, resulting in state S0.

[Axiom 48]
**Cancel()** of initiator cfact state S2:

\[
\text{Actor.Cancel}(i) : \forall i \in I_i
\begin{aligned}
& \text{Actor.Init}[i] \\
& \Rightarrow \text{Actor.Init}[i]
\end{aligned}
\]

For cfact state S2, a decline has been issued and after a **Cancel()** the initiator issues a quit, resulting in cfact state S5.
5 Design of a Dynamic Logic Model for the Ψ Theory

[Axiom 49]

Cancel() of initiator cfact state S3:

\[
\forall i \in \{\text{ActorInit}\} : \begin{cases}
Rq = 0 \\
Pm = 1 \\
Dc \cdot Qt = 0 \\
St, Ac, Rj, Sp = 0
\end{cases} \Rightarrow \begin{cases}
Rq = 0 \\
Pm = 1 \\
Dc = 0 \\
Qt = 1 \\
St = Ac = Rj = Sp = 0
\end{cases}
\]

For cfact state S3, a promise has been issued and after a Cancel() the initiator issues a quit, resulting in cfact state S4.

Cancel() of initiator cfact state S4:

\[
\begin{cases}
Rq = 0 \\
Pm = 1 \\
Dc = 0 \\
Qt = 1 \\
St, Ac, Rj, Sp = 0
\end{cases}
\]

[Cfact S4]

The initiator has already issued a quit after a promise; no state change is required, resulting in cfact state S4.

Cancel() of initiator cfact state S5:

\[
\begin{cases}
Rq, Pm = 0 \\
Dc, Qt = 1 \\
St, Ac, Rj, Sp = 0
\end{cases}
\]

[Cfact S5]

The initiator has already issued a quit after a decline; no state change is required, resulting in cfact state S5.
5 Design of a Dynamic Logic Model for the Ψ Theory

[Axiom 50]

Cancel() of initiator cfact state S6:

\[
\forall i \in \text{Actor.Init}[i] \Rightarrow \text{Actor.Init}[i] \quad \begin{cases} 
Rq = 0 \\
Pm = 1 \\
Dc, Qt = 0 \\
St = 1 \\
Ac, Rj, Sp = 0 
\end{cases} \Rightarrow \begin{cases} 
Rq := 0 \\
Pm := 1 \\
Dc := 0 \\
Qt := 1 \\
St := 1 \\
Ac := Rj := Sp := 0 
\end{cases}
\]

For cfact state S6, a promise has been issued and after a Cancel() the initiator issues a quit command, resulting in cfact state S8.

[Axiom 51]

Cancel() of initiator cfact state S7:

\[
\forall i \in \text{Actor.Init}[i] \Rightarrow \text{Actor.Init}[i] \quad \begin{cases} 
Rq = 0 \\
Pm = 1 \\
Dc, Qt = 0 \\
St = 1 \\
Ac = 0 \\
Rj = 1 \\
Sp = 0 
\end{cases} \Rightarrow \begin{cases} 
Rq := 0 \\
Pm := 1 \\
Dc := 0 \\
Qt := 1 \\
St := 1 \\
Ac := 0 \\
Rj := 1 \\
Sp := 0 
\end{cases}
\]

For cfact state S7, a reject has been issued and after a Cancel() the initiator issues a quit, resulting in cfact state S10.
Cancel() of initiator cfact state S8:

\[
\begin{align*}
\text{Rq} &= 0 \\
\text{Pr} &= 1 \\
\text{Dc} &= 0 \\
\text{Qt} &= 1 \\
\text{St} &= 1 \\
\text{Ac, Rj, Sp} &= 0
\end{align*}
\]

[Cfact S8]

The initiator issued already a quit after a state; no state change is required, resulting in cfact state S8.

[Axiom 52]

Cancel() of initiator cfact state S9:

\[
\begin{align*}
\text{Actor . Cancel()} & \\
& \forall i \in \text{Init}\{ \}
\end{align*}
\]

\[
\begin{align*}
\Rightarrow \text{Actor . Init}[i] \left( \begin{array}{c}
\text{Rq, Pr, Dc, Qt} = 0 \\
\text{St, Ac, Rj} = 0 \\
\text{Sp} = 1
\end{array} \right)
\end{align*}
\]

For cfact state S9, a stop command has been issued and after a Cancel() the initiator restores to the default cfact state S0.

Cancel() of initiator cfact state S10:

\[
\begin{align*}
\text{Rq} &= 0 \\
\text{Pr} &= 1 \\
\text{Dc} &= 0 \\
\text{Qt} &= 1 \\
\text{St} &= 1 \\
\text{Ac} &= 0 \\
\text{Rj} &= 1 \\
\text{Sp} &= 0
\end{align*}
\]

[Cfact S10]
The initiator has already issued a quit after a reject instruction and hence no state change occurs.

**Cancel()** of initiator cfact state S11:

\[
\begin{align*}
Rq, Pm, Dc, Qt & = 0 \\
St & = 0 \\
Ac & = 1 \\
Rj, Sp & = 0
\end{align*}
\]

[Cfact S11]

The transaction has already completed; as the transaction is accepted and final, no state change occurs.
5.6.3. **State Transitions overview with Cancel Patterns**

Figure 5.10 Cfact State and Transition spaces with cancellations.
A graphical representation of all allowed cfact states – S0, ..., S11 – in addition to standard and cancellation state transitions is shown in figure 5.10. The same remark about matrix representation for fig. 5.9 applies here too.

Figure 5.10 is identical to figure 5.9 where the additional cancellation state transition functions have been added. It is clear that no new states have been created. There are only more optional transitions.

The cancellation pattern, triggered by an actor, may induce also the upward and downward rollback propagation for the executor and initiator roles. The details of these mechanisms are discussed in section 5.2 and validated through simulations in Appendix I. The absence of anomalies, represented by the consistency condition of the C4-ness criteria, is also discussed.

5.7. Application of Business Rules to Transaction Patterns

In the present section, the transaction pattern is derived in compliance with the Ψ theory. However the reality of production may dictate the necessity for additional constraints to be applied to actor decisions. An example is the requirement that one production item should be produced and accepted before the production of another item can commence. Constraints such as these cannot be implemented in the transaction pattern because they are not part of the Ψ theory. They are specified in the DEMO action model and are to be implemented in the action model processor (section 6).

5.8. Conclusions and Results

The second and third cardinality law have been applied to the Ψ theory tuple, following the GSDP-MDE approach, for the dynamic behavior of Ψ models. The 52 axiomatic state transaction specifications specify all 12 valid transaction states and all valid transaction state changes for each transaction in any, unlimitedly large aggregated construction model. For each specific transaction, there is total compliance with the transaction axiom of the Ψ theory. These specifications are, together with the specifications of section 4, a complete specification for a software implementation.

Validation provides the conclusion that all phenomena that may be observed in reality are supported:

Absence of anomalies

The absence of anomalies, represented by the 'consistency' condition of the C4-ness criteria. This is discussed in section 3.4.8. In section 6.4 the C4-ness qualities of DEMO models is supported by additional reasoning.
Major and minor loops
Transaction systems may exhibit endless loops in transaction cycles that may span more than one transaction. These are normal phenomena and are observed in the real world.

Propagation mechanisms
The transaction state changes include several mechanisms of propagation:

i.) Transaction propagation for a successful (as intended by the modeler) execution, in which for each transaction a request – promise – state – accept sequence is followed.

ii.) An upward transaction rollback mechanism, induced in situations when at a lower level a pfact cannot be delivered.

iii.) A downward transaction rollback mechanism, induced to terminate transaction processing.

Cancellation patterns
The cancellation patterns, triggered by an actor, may induce also the upward and downward rollback propagation for the executor and initiator roles. The details of these mechanisms are discussed in detail in section 6 and validated through simulations in Appendix I. The implementation of cancellation and various choices to be made requires further investigation.

The results of this section, in addition to section 4, are part of the programming specifications of the software Ψ processor and DEMO processor.
6. **Ψ Processor Validation, Verification and Quality Assessment**

*Ignorance is Bliss: So why aren’t more people happy?*

Alfred E. Neuman

The American philosopher Alfred E. Neuman advises us to enjoy the bliss of ignorance while it lasts, at least until any software validation and verification activities start.

**Abstract.** In this section the behavior and role of the Ψ processor as a control system for enterprises, based on work of Guerreiro, S. [2011, 2012], is investigated. It is shown that the Ψ processor is a descriptive information system, giving complete information about the current state of the enterprise, whilst also being a prescriptive information system that enforces model compliance. From this perspective it is a ‘classic’ control system, which has been validated by Guerreiro. It is necessary to examine the behavior of the Ψ processor in order to investigate its validity in addition to several quality issues, as formulated in section 3. The first quality topic is the validation of phenomena observed in reality against the simulated behavior of the Ψ processor, which constitutes an assessment of its ontological truthfulness and completeness. The second topic is the assessment of DMOL language quality concerning the minimization of expressiveness and programming itself, as well as errors related to programming. The third quality topic is the establishing of additional support for the C4-ness claim of the Ψ theory and of enterprise ontology. The fourth topic is about defining the criteria that should be met for information systems that capture an enterprise.

6.1. **The Ψ processor Role Based Control System for Enterprises**

Enterprise information systems should capture the operation of an enterprise in an appropriate and truthful way. The term ‘capture’ does not merely refer to description but also to prescription. Prescription implies a control system, as found in industry and many appliances. The purpose of control systems is to prescriptively influence the behavior of some system. They are often also a descriptive information systems that provide data about the state of the system under control. Their key elements are:

1. the target system under control that should exhibit some desired functional behavior;
2. the control system connected in some working way to the target system;
iii.) some reference or a model that represents a desired state, value or behavior of the target system.

The control system takes the following actions:

i.) measurements of the current state, values or behavior of the target system;

ii.) calculation of controlling actions based on the measurements, optionally older values, and the reference model;

iii.) enforcement of the target system to act according to the calculated results.

In this section the operation of the DEMO processor as a role-based control system, acting as a controlling virtual actor, is investigated.

6.1.1. Fine-grained Role Based Access Control for Enterprises

A high quality design of an ideal enterprise for some production is adequate, but not good enough. The large and unmanageable chaos in the communication of any enterprise of more than a few actors demands a control system with adequate calculation capabilities. Guerreiro extensively investigated the control of enterprises against DEMO enterprise models using the $\Psi$ processor [Guerreiro, 2011]. The roles of all actors within the enterprise are controlled at runtime; hence, the term ‘role based access control’ (RBAC) of actors. The term ‘fine grained’ refers to the level of detail of control; each elementary act resulting in a fact is controlled for each individual actor. The overall enterprise governance, the set of rules to which all actors have to comply, is similarly controlled.

The control system monitors the current state of the enterprise and compares it to a model of the enterprise, calculating the allowed actions of the actors and controlling the behavior of each actor. Controlling the allowed behavior of actors is done by providing each of them with a set of allowed options to act in communication. These communication acts result in communication facts. Any actor action that is not in line with the enterprise model is prohibited. The investigation of RBAC by Guerreiro is essential in research on the DEMO processor. While this research is about construction and operation of the $\Psi$ processor, the white box model, RBAC validation encompasses the detailed black box validation of the $\Psi$ (or DEMO) processor, its internal correctness.

6.1.2. A Control System for Communication and Coordination

In order to design a control system, the relation between the actors, together with their communication, and the enterprise model should be investigated. The enterprise model is a high quality design of the desired enterprise, developed using the DEMO methodology.
6. $\Psi$ Processor Validation, Verification and Quality Assessment

It specifies the communication between two actors precisely, in detail – in a fine-grained manner – according to the transaction axiom, for any actors that are involved in a transaction.

It is assumed here that the $\Psi$ model is unlimitedly large, with an unlimitedly large number of actors and transactions, with full compliance to the three axioms of the $\Psi$ theory. Applicable to any actor within the configuration, the scheme of figure 6.1 represents a fragment of this large model.

Figure 6. 1. Aggregated formal model with transactions and actors (top). Part of the model (bottom) with only communication and actors represented.

Within the model (top) shown of figure 6.1, there is an unlimited number of parent actors and {grand-}parent transactions for actor A0. Similarly, for actors A2A and A2B there is for each an unlimited number of {child-}child-actors and transactions. The situation for

---

87 On one side the calculation of the model is completely deterministic, the EIS and the controlled actors cannot deviate from the model. On the other side Demo models allow actors to choose, they may issue a promise or a decline, an accept or a reject, etc. In this way there is full freedom for the actor to act, even capricious or with any lack of responsibility of truth. The model under execution controlling an enterprise is in this way not a deterministic system.
actor A1, which represents any actor within any imaginable Ψ model, is investigated as follows.

Any actor A1 has one and only one executor role, except the root initiator, which is typically self-initiating. Any actor A1 has 0, 1 or n initiator roles, with other actors taking executor roles. In figure 5.1, for actor A1 two initiator roles are shown. In the investigation of actor A1, any {grand-}parent transactions for A0 and any {child-}child transactions for A2A and A2B as are irrelevant and can be discarded. In an inductive way figure 6.1 represents the situation for any actor in a large Ψ model.

While the top scheme of figure 6.1 shows complete transactions that are relevant for actor A1, we are only interested in communication and control and discard the p-world. The result, a generic model with only actors and communication, the a-world and c-world according to the operation axiom, is shown in the bottom model\(^88\).

### 6.1.3. The Controlling Virtual Actor

From the viewpoint of actor A1 there is only communication with actors A0, A2A and A2B, with both initiator and executor roles. Since a control system is needed, it must be one that not only acts as any actor but also enforces compliance of communication to the transaction axiom.

---

\(^{88}\) This model is not a correct DEMO model. For convenience we applied the Organization Construction Diagram legend of Appendix III.
The control system is a virtual actor that handles all other actor roles, A0, A2A and A2B, that communicate with actor A1 and that enforces compliance to the model as well as compliance to the transaction axiom patterns for each transaction.

The communication between the virtual control actor with all actors is shown in figure 6.2. The virtual control actor takes the roles of three actors, A0, A2A and A2B, for the communication with actor A1. Similarly the virtual control actor takes the role of actor A1 with the other actors A0, A2A and A2B. The distinction between executor and initiator is lost since both roles are taken.

**The virtual control actor as an agent for all actor roles**
The active control for transaction C1 is shown in figure 6.3 (left). In the scheme on the right the communication is shown in more detail. The virtual control actor issues a message to actor A1 with a question (Q) and valid options (O) to choose from. The question informs actor A1 in detail which communication state of which transaction is at stake and asks for a decision; actor A1 is now fully informed. The message ‘C1 D’ from actor A1 to the virtual control actor contains the decision (D) made by actor A1 for the allowed options.

An example is shown in figure 6.3. Actor A0, through the virtual control actor, just issued a request to actor A1, a question, for transaction T1. The allowed options for actor A1 are either a promise or a decline. Any other option is illegal and prohibited.

![Diagram](https://example.com/diagram.png)

**Figure 6.3.** Detailed Communication between the virtual control actor and actor A1.

The behavior of the virtual control actor is, in DEMO terminology, an ‘agent’ for all actor roles. I.e. any actor ‘sees’ only the virtual control actor; the other actors are hidden behind the virtual control actor.

6.1.4. **The Model Calculation by the Virtual Control Actor**
The virtual control actor needs to have full knowledge of the \( \Psi \) model and of the current state of any transaction. By applying the transaction transition space axioms of section 4 to
the current transaction state space, the virtual control actor calculates the communication with each actor. Note that the virtual control actor may take multiple active communication acts for multiple transactions with any actor at the same time.

**Psi Processor Communication with Actors**

The virtual control actor is implemented by the Psi processor that executes a Psi model, as shown in figure 6.4. For each actor, the set C* of actor roles represents their communication with the Psi processor. The distinction of any role as executor or initiator is lost, since any actor may take several roles. However, this knowledge is calculated by the Psi processor for each individual transaction.

![Diagram](image)

**Figure 6.4.** A simplified representation of figure 6.2. Each actors communicates exclusively with the virtual control actor, the Psi processor executing a Psi model.

The connection between the Psi processor and the Psi model is bi-directional, to express that after each communication act the current state of the Psi model under execution is modified accordingly. After each single communication act, a communication fact is produced. The state space of the whole Psi model is modified and for all transactions, the overall communication is recalculated.

Typically this communication is asynchronous; the most appropriate medium for this type of communication is email. It is essential that each physical actor in the enterprise is fully informed about any relevant transaction, in order that they are able to take justified decisions that are fully compliant to the enterprise Psi model.
6. \( \Psi \) Processor Validation, Verification and Quality Assessment

6.2. Validation by Simulation

Based on the specifications of sections 4 and 5, the \( \Psi \) processor has been implemented in software and subjected to model simulation. There are three reasons as to why the analysis, the construction of the \( \Psi \) processor automaton and the extensive modeling and simulation of the production rules and transaction patterns, are necessary (Appendix I). These reasons are outlined as follows.

The first reason is that extensive simulation behavior must be validated against phenomena in reality, further supporting the claim of ontological truthfulness and completeness of enterprise ontology. This adheres to design science theory [Hevner, 2004]; after the build process of an artifact an evaluation process is mandatory. All phenomena observed in reality, such as the successful execution of aggregated transactions, the two rollback propagation mechanisms, collision handling and cancellation mechanisms, are validated (and found to be correct).

The second reason is that the transaction pattern at the elementary level – the single transaction model – is consistent with the patterns in unlimitedly large transactional systems, while maintaining soundness, lack of ambiguity or deadlocks. The 52 axiomatic state transaction specifications (section 4) specify all 12 of the valid transaction states and state transitions for any unlimitedly large aggregated actor-transaction diagram. At the same time for each specific transaction there should be total compliance with the transaction axiom. Similarly, for the construction of models, the production rules of the \( \Psi \) processor (section 4.3.8, 4.3.9) are validated.

The third reason for which validation is necessary is that extensive formal analysis on paper and programming a software automaton are both complex and subject to human programming errors. As such, manual validation on paper is inadequate. The closed loop of validation against reality and the enterprise ontology axioms is good but not yet good enough.

Verification using Temporal Logic Simulations

The previously described methods cannot deliver guaranteed completeness of all possible states and state transitions. Temporal logic simulation\(^{89}\) is a brute force validation that may provide verification with guaranteed completeness. This is a future research topic. It will test truthfulness, soundness, anomalies, completeness, deadlocks and compliance to the transaction axiom for unlimited large models.

\(^{89}\) Temporal logic simulation is a method to describe and check systems that represent propositions that involve (also) discrete state changes over time, such as in this research. It includes model checking for correctness and completeness. Important contributors are Mordechai Ben-Ari, Zohar Manna, Amir Pnueli, and Hans Kamp.
Verification of Truthfulness and Soundness

The validation process concerns a number of important phenomena observable in the real world. Extensive simulations have been carried out in Appendix I and simulation logging can be validated in detail against the state transition axioms of section 4. There are seven key issues involving this process. These phenomena, outlined as follows, are compliant with those we observe in reality and support the claim of ontological truthfulness and apparent lack of anomalies.

Absence of Anomalies and Deadlock Situations

A deadlock situation is one in which a certain transaction state (or set of states) cannot be exited. This may be because two or more conditional transitions are waiting for the others’ outcomes. Soundness is defined by the possibility to achieve any imaginable state, starting from any imaginable state. There is one exception to this: once a transaction has been completed (accepted = true), this state can no longer be reversed. Deadlock conditions do not exist theoretically in the $\Psi$ models, because there are no interdependencies and at the level of a single transaction the absence of deadlock conditions can indeed be shown. Validation of present constructions (Appendix I) did not produce any deadlock situations. In reality, in any enterprise there are many possible scenarios in which interdependencies do exist, such as when one condition is waiting for another. Deadlocks may occur in these instances and therefore any models constructed by way of the DEMO processor, with Action Model rules, should exhibit this deadlock potential. The absence of anomalies also implies model completeness: that any model complies completely with the $\Psi$ theory axioms.

Major and Minor Cycles

In section 5.4, figure 5.8, an example of a repeating cycle, a loop, is shown. If a transaction step is refused (such as in the case of request-decline, or of state-reject) for some unknown reason – a reason at a higher abstraction level – the transaction pattern can be repeated an unlimited number of times for the same pfact (or set thereof) according to the transaction pattern axioms. While this is formally correct in that it is derived directly from the $\Psi$ axioms, it may not make much sense in reality; human actors will not repeat transaction patterns endlessly without reason in this way. Nevertheless, it is a possibility that these loops do occur in reality.

Request-Decline and State-Reject Minor Loops

The first minor loop that may occur is the single level request-decline loop, following axioms 6 and 7 (section 5.5.3, 5.5.4). An initiator and his executor may repeat a request-decline cycle repeatedly for any reason until this reason disappears. The same applies for the state-reject minor loop, axiom 25 and 26 (section 5.5.7).
Transaction Downward and Upward Rollback Phenomena

In any state based production system, it must be possible to reverse or roll back transactions at any time and for any reason. If a transaction rolls back, any related transactions should also roll back recursively and optionally until all transactions are rolled back. Transaction roll-back capabilities are investigated in Appendix I, Simulations.

This is a common situation in reality in the event that the assembly of some product fails for lack of any constituent of the product; if this occurs then all other transactions, finalized or not, must roll back as shown in figure 6.5. The two recursive rollback mechanisms of the transaction pattern are specified in section 5.5.8. Simulation of the $\Psi$ processor in Appendix I validates these mechanisms in detail.

Figure 6.5. Transaction Rollback mechanisms.

Figure 6.5 shows a hierarchical tree of transactions; the actors and cfacts are omitted for simplicity. First, the cycles of phase I are executed according to the normal request-promise cycle for the transaction chains, denoted in figure 5.5 as T00-T10-T20-T30 and T00-T11-T23-T33. At transaction T30 of phase II, the state-accept cycle is completed; T30 is accepted and the pfact is available. Simultaneously in phase II of T33, a state-reject-quit cycle is complete, and hence the transaction T33 fails in production. At this juncture the upward rollback stop-decline-quit cycle is completed as shown by phase III. Once this cycle arrives at T00 then phase IV commences, in which the downward rollback quit-quit-
acknowledge-quit cycle is executed until this cycle arrives at T30. The overall production of T00 fails due the failure of T33. At phase V, the promise for T00 is declined. A repeated request for T00 is possible of course.

**Soundness Criterion**

The soundness criterion states that any imaginable model in any state can be reached from any other model in any state. From any model, any new model can be constructed by using the DelPfact() and LinkPfact() functions, assigning new attribute values; any transaction state can be reached in this way unless a transaction has been completed.

6.3. **DMOL Language Quality Assessment**

This section deals first with the quality evaluation of the DMOL grammar at the real world semantic level: the question of if – and to what extent – the DMOL language and its primitives are suitable to express models clearly, if they are valuable and if ambiguity to all stakeholders is eliminated. In section 3.4 it is shown that the three cardinality laws of the GSDP-MDE methodology are mandatory in order to obtain ontological clarity and to eliminate construct overload and programming with resulting errors. It is also demonstrated that minimized expressiveness supports optimal model validation. This ontological clarity and lack of construct overload for any DMOL model is verified by investigation of the 1:1 cardinality between instances of the tuple elements of tuple 1 and the primitives and constructs in the DMOL language, as shown in figure 3.6. If this validation is successful, the cardinality laws have been applied properly for the DMOL metamodel, then C4-ness quality of DMOL models can be claimed (section 3.4.8).

Second, it must be shown that any constructed DMOL model using the production rules is a verified Ψ model. This is a quality requirement of the Ψ processor, the correct implementation of the DelPfact() and LinkPfact() functions (section 3.2.10).

The $m : M : S = 1 : 1 : 1$ relation, cardinality law 1, should be investigated and verified beginning from the tuple:

\[
\text{< Pf, Pcom, Cf, CP, Act, Executor, R, CR, Attribs, Attr >} \quad \text{[DEMO Tuple]}
\]

This evaluation is simply shown by investigation of an example which is compliant to the grammar specification in section 4.4.2. This example of figure 4.11 has been manually edited; an example of an action model rule has been substituted.

\[
\text{<DMApp>DEMOApplication} \\
\text{<PFactChilds>} \\
\quad \text{<PFact>} \\
\quad \quad \text{Identifier = "Tyre" Label = ""}
\]
<CFact>
Identifier = "" Label = ""
    Requested = "false" Promised = "false"
    Stated = "false" Accepted = "false"
    Declined = "false" Quited = "false"
    Rejected = "false" Stopped = "false"
</CFact>
<ActExec>
    Identifier = "Tyre Factory" Label = ""
</ActExec>
<ActInit>
    Identifier = "Tyre Customer" Label = ""
</ActInit>
<PFactChilds>
    <PFact>
        Identifier = "Rubber" Label = ""
        <CFact>
            Identifier = "" Label = ""
            Requested = "false" Promised = "false"
            Stated = "false" Accepted = "false"
            Declined = "false" Quited = "false"
            Rejected = "false" Stopped = "false"
            <ActionRules>
                OnRequested
                if Infologic( Opcode, ParamList ) == true then
                    SetStatus( "", Decline ) /* Executor declines */
                else
                    SetStatus( "", Promise ) /* Executor promises */
                fi
            no
            </ActionRules>
        </CFact>
    </PFact>
</PFactChilds>
</DMApp>

Figure 6.6. Example of a DMOL file with an action model rule.

Investigation of this example code delivers:

**Pf**: a set of pfacts, the production base.

Each Pf instance, with an identifier and a label, is represented by:
The relation \textbf{Pcom} between two Pfacts, each with their Identifier and their Label, is represented by:

\begin{verbatim}
<PFact>
  Identifier = "Tyre" Label ="
</PFact>

<PFactChilds>
  <PFact>
    Identifier = "Rubber" Label ="
  </PFact>
</PFactChilds>
\end{verbatim}

\textbf{Cf}: a set of cfacts, the coordination base, representing the state space.

Each \textbf{Cf} instance is represented by:

\begin{verbatim}
<CFact>
  Identifier = "" Label ="
  Requested = "false" Promised = "false"
  Stated = "false" Accepted = "false"
  Declined = "false" Quited = "false"
  Rejected = "false" Stopped = "false"
</CFact>
\end{verbatim}

\textbf{CP}: a set of relations between Cfacts and Pfacts

Each \textbf{CP} relation between a Pfact and a Cfact instance is represented as:

\begin{verbatim}
<PFact>
  Identifier = "Tyre" Label ="
</PFact>

<CFact>
  Identifier = "" Label ="
  Requested = "false" Promised = "false"
  Stated = "false" Accepted = "false"
  Declined = "false" Quited = "false"
  Rejected = "false" Stopped = "false"
</CFact>
\end{verbatim}

\textbf{Act}: a set of Actor roles,
Each Actor is represented as:

```xml
<ActExec>
  Identifier = "Rubber Factory" Label ="
</ActExec>
```

and for the ROOT Pfact there is also:

```xml
<ActInit>
  Identifier = "Tyre Customer" Label ="
</ActInit>
```

Executor: a set of relations between a Pfact and an Actor

Each Executor relation is represented as:

```xml
<PFact>
  Identifier = "Rubber" Label ="
  <ActExec>
    Identifier = "Rubber Factory" Label ="
  </ActExec>
</PFact>
```

R: a set of rules and dependencies, the Rule Base.

Each R, Rule instance is composed of a set of type Line as specified in section 3.4.3.

```plaintext
OnRequested
  if Infologic( Opcode, ParamList ) == true then
    SetStatus( "", Decline ) /* Executor declines */
  else
    SetStatus( "", Promise ) /* Executor promises */
  fi
no
```

CR: a set of relations between Cfacts and Rules.

Each CR instance is represented as:

```xml
<CFact>
  Identifier = "" Label = ""
  <ActionRules>
```
6. Ψ Processor Validation, Verification and Quality Assessment

These are shown in this example of a Pfact:

```
<PFact>
  Identifier = "Rubber" Label = ""
</PFact>
```

All the sets of the tuple have been verified and found to comply.

**Conclusions**

The verification demonstrated here shows that, due to the \( m : M : S = 1 : 1 : 1 \) cardinality between concepts and entities and their constructs in the DMOL language for any imaginable Ψ model, there is always one and only one corresponding DMOL representation. By application of the production rules on a verified Ψ model, the resulting constructed Ψ model is hence also verified. By repeated application of the production rules, any verified Ψ model can be constructed from any other verified Ψ model. Any constructed Ψ model corresponds with one and only one instance in reality. There is ontological clarity and no construct overload. The minimized expressiveness quality eliminates the construction of models that cannot exist in reality. These are high quality language criteria, in line with section 3.4.1. Using the results of section 3.4.8 we may conclude that DMOL models have C4-ness quality.

**6.4. Support for the C4-ness claim of the Ψ Theory**

The formal correctness of DEMO models is formulated by the C4E\(^{90} \) criteria, which is a key quality standard that is claimed for models. In section 3.4.8 the C4-ness qualities are defined. Any lack of C4-ness is related to improper application of the three cardinality laws of the GSDP-MDE methodology. In section 6.3 the C4-ness qualities of DMOL models has been verified. The following reasoning further supports the existing C4-ness claim [Dietz, 2006] of the Ψ theory.

---

\(^{90}\) C4E is composed of C4-ness plus “E-ness”. C4-ness referring to Coherence, Comprehensiveness, Consistency and Conciseness (section 3.4.8). “E-ness” refers to “Essential”, a quality criterion for DEMO models which implies that only ontological (essential) transactions are shown in a DEMO model, section 3.6.4.
Coherence
Coherence refers to the ‘semantic meaningfulness of the symbols and their relations from
every perspective’. This means that the four DEMO aspect models (section 3.8),
representing an enterprise from every perspective, must relate to the same unique DMOL or
NLDLMOL model as well as to the same enterprise under investigation. It is straightforward
to (re)construct the three unique DEMO aspect models (the construction, action and process
models) from a (single) given DMOL model. For the reconstruction of the state model from
DMOL the entire population of a DMOL model in all possible states is required. This
supports the claim that all four DEMO aspect models are coherent and can be represented
by a single DMOL model.

Figure 6.6. Real World phenomena, enterprises, related to conceptualizations of enterprise ontology,
related to modeling languages and the applied cardinality laws, derived from figure 3.5 and 3.7.

In section 7.1.3 is shown that one and only one DMOL model can be constructed from a set
of DEMO models and that from any DMOL model one and only one set of DEMO models,
the original DEMO models can be reconstructed, without any loss or anomalies.
It is shown in section 6.3 that the DMOL metamodel has C4-ness qualities. From this
follows that the DEMO model metamodels must have C4-ness quality.
Figure 6.6 shows that the metamodels of DMOL and DEMO aspect languages are isomorphic to enterprise ontology, with 1:1 cardinality.

**Essence and Distinction**

As stated in section 3.6, the distinction axiom of enterprise ontology is being used by the human modeler to separate essential transactions from non-essential transactions. This is done before the construction of any models and is hence outside the scope of this research. Hence no claim can be made about the "E" of the C4E-ness claim of the theory of enterprise ontology.

### 6.5. Information Systems capturing an Enterprise

This section discusses the relation between an information system and an enterprise, also called Business-IT alignment. The IT governance institute\(^9\) defines "alignment is the capacity to demonstrate a positive relationship between information technologies and the accepted financial measures of performance". In the mission statement of this research, it is stated that information systems should capture the operating enterprise in a truthful and appropriate way. In light of this, the following three essential issues apply.

The first issue is the prescriptiveness versus descriptiveness of an information system. In section 6.1 it is shown that the Ψ axioms and hence enterprise ontology and the DEMO methodology are a prescriptive system, not just a descriptive specification in which actors can deviate from the axiomatic specifications. It enforces all actors to comply with the allowed state transitions. In addition business rules are implemented in the action model processor (section 7.3). This has also been extensively investigated and validated for role based access control mode enforcement (RBAC) in runtime transactions for enterprises [Guerreiro et al., 2011; Guerreiro 2011]. At design time, the developer may decide to enforce the business rules in a rigid way or to leave the actor to overrule a business rule using his superior cognitive capabilities. At the business operating level, it implies that in this way staff is enforced to comply with business processes. However, the rollback capabilities and cancellation patterns allow an actor to 'break out' at any time for any unforeseen reason. It is recommended in a general way, based on professional experience, to allow an informal exception handling based on common sense and responsibility afterwards. This is easily implemented in the action model processor.

The second issue is that a prescriptive control system must always be perfectly deterministic and with lack of anomalies such as deadlocks and ambiguity.

The third issue is the relation between transaction types versus transaction instances. The transactions at the Ψ processor level are transaction instances, as opposed to transaction

---

9 The IT institute (http://www.itgi.org/), 3701 Algonquin Road, Rolling Meadows, IL 60008 USA.
types in the DEMO models. We model the enterprise resulting in a set of model types, typical of a library. As shown in the ontological parallelogram (section 3.3.3) in the production for each product instance a new occurrence of that model must be instantiated, typically by a copy operation. If a transaction involves multiple identical pfacts – ‘do this n times’ – then for each pfact a separate transaction instance must be specified.

6.6. Conclusions and Results

The following results have been achieved:

i.) Simulations and pragmatic validation of correctness have been carried out, described in Appendix I.

ii.) The compliance of the execution of Ψ models against the Ψ theory axioms has been executed in detail.

iii.) Perfect compliance and no anomalies have been found so far. However, more rigid temporal logic validation is required, one of the future research topics.

iv.) Large-scale enterprise models in simulation exhibit all the transaction phenomena that can be observed in enterprises in reality. This applies especially to nested transaction rollback phenomena. This validation delivers further support for the claim of ontological truthfulness and completeness of enterprise ontology. In addition, this supports a claim of formal correctness of the specifications and the software implementation.

v.) Reasoning is supplied for the claim of C4-ness of DMOL models and for DEMO models.

It is concluded that the Ψ processor is an information system 'in the traditional sense', because information about the enterprise is provided to the "outside world" of stakeholders and external information systems. The Ψ processor is also a runtime role-based control system for enterprises because information is provided "within" the enterprise to the individual actors. This enforces model compliance to each individual actor of the enterprise (section 6.1).
6. \Psi\text{ Processor Validation, Verification and Quality Assessment}
PART IV

DEMO Processor Design and Construction
7. Demo Processor Design and Construction

Homo Ergaster, the “workman” is an extinct human species, widely accepted as our ancestor, that lived between 1.3 and 1.8 million years ago.

H. Ergaster applied the engineering science by constructing stone tools to meet certain functional requirements, cutting meat. He was an engineer and shows that engineering is one of the oldest professions of mankind.

Abstract. Currently, DEMO models are being developed and used as an optimal point of departure for any implementation of an enterprise. To use these DMOL models for the purposes of monitoring and controlling an enterprise, we need a software engine: the DEMO processor. The DEMO processor is an extension of the Ψ processor with enhanced capabilities to operate in a professional modeling and production environment. This design is the first design cycle step, based on experience with an earlier GSDP-MDE like software development system (Appendix V) and common engineering creativity. Further design cycles have to be made. The main purpose of this section is the first design step and to show the feasibility of the DEMO processor.

7.1. Introduction to the DEMO Processor

In sections 4 and 5, the subject of interest was the design of a finite state machine with full compliance to the Ψ theory, as formulated by the second research question (section 1.3.2). In Appendix II, the construction of the Ψ processor in software is described, while in Appendix I simulations and verifications using the Ψ processor have been carried out. These results show that the Ψ theory is a ‘true’ ontology in the formal sense: a formal specification, as discussed in section 2.4. This conforms to the requirements of high quality software specifications, constructs and primitives, as formulated in section 2.4.4 and verified in detail in section 5. Section 5 also accounts the validation of various phenomena observed in reality versus the behavior of the Ψ processor, and these are found to be in support of the truthfulness claim. To summarize, the Ψ theory is a formal and truthful ontology that describes enterprises in operation in an appropriate way and that enables the design of a high quality software engine, the Ψ processor.

This section deals with the question of if and to what extent we can construct a DEMO processor that supports the construction of the DEMO aspects model, and that simulates or
executes these models for modeling and production environments. This theme is formulated succinctly in the third research question:

*How to design the DEMO processor, based on the \( \Psi \text{ processor, that supports all four DEMO aspect models?} \)*

The DEMO methodology (section 2.5) delivers four DEMO aspects models. These coherent models specify much more about an enterprise than the \( \Psi \) theory. The present section discusses the design of the software DEMO processor as based on the \( \Psi \text{ processor.} \)

The DEMO processor is defined as an automaton that is derived from the three axioms of the \( \Psi \text{ theory.} \) This automaton is in fact a set of cellular automata, in which each automaton controls a specific transaction of an unlimitedly large DEMO model. The DEMO processor executes a DMOL model. The DMOL model is a representation – as close a representation as possible – of the four DEMO aspect models. The DEMO processor supports the ability to create and to modify formally correct and verified DEMO models. It also enables the execution or simulation of the dynamic behavior of a DEMO model as well as allowing for rendering of the data to construct associated DEMO aspect models. The DEMO processor must be able to perform a number of tasks and meet a set of requirements. Each of these is explained and the motivations for their existence examined in the present section. The relation between the four DEMO aspect models and the DEMO processor is as follows.

### 7.1.1. Engineering Methods Applied

The purpose and the application of the DEMO processor are described in sections 7.1, 7.1.2 and 7.1.3. A number of design requirements, required capabilities, considerations and choices made are described in section 7.2. The solution space is determined and limited by the many requirements and constraints for abstractions and minimized expressiveness. The translation of functional to construction is so far informal, “creative software engineering”. The design is so far incomplete, for the action model processor (section 7.3.1) a primitive instruction set is proposed and some examples are given. This design is however also based on experiences of an earlier GSDP-MDE programming system, of which the development started in 1994 and that went into professional production in 1997 (Appendix V). This design is a reiteration of that mature GSDP-MDE system, which provides some confidence that this design so far is a “good” first design cycle. The design results so far are still the first part of the first design cycle, with yet minimal validation and testing. Much further research has to be done, formulated in section 8.9. With good confidence the feasibility of the DEMO processor has been shown.
7. Demo Processor Design and Construction

7.1.2. DEMO Models

DEMO models are models represented by ordered and related symbols that:

i.) are specified by formal graphical languages;

ii.) are intended for the purpose of shared understanding and reasoning by human stakeholders;

iii.) specify that which is essential, whilst discarding anything that is specific to implementation or that what is not captured by enterprise ontology;

iv.) express these features in specifications composed of (i) classes, result types, fact types with occurrence and unicity laws; (ii) actors, productions and transactions and (iii) their relations with regards to the static state and dynamic transition state spaces.

As yet, there is still no trace of any such implementation since the models are ontological, any implementation is still abstracted according to the distinction axiom (section 3.6.4). For any operational implementation, it is argued that the DEMO methodology is highly suitable for further design and execution of enterprises. This is achieved by recursive and iterative redefining the scope of essential and non-essential transactions and including infological and datalogical transactions. This method, fine-grained DEMO, delivers very large and detailed DEMO models, far too complex for human reasoning, that specify in detail enterprises in operation. For these models a DEMO processor is needed.

7.1.3. The DEMO Processor

The DEMO processor is an automaton (comprising a set of automata or cellular automata) that enables the construction of DEMO models in both static and dynamic spaces.

It allows for the construction of any imaginable and verified DEMO model, whilst excluding the construction of anything that is not a DEMO model.

It is intended for the purposes of (i) constructing and simulating models for model validation; ii) enforcing enterprise operation to be compliant with a specific model and (iii) delivering complete state information about the enterprise, which defines it as an enterprise information system.

It should handle DEMO models as a totally model independent software system.

The first implementation of DEMO derived IT systems has been done by Mulder [Mulder, 2007]. A specific case study (SGC) has been supported by an implemented IT system. This case shows the benefits of IT systems developed from DEMO models.

Here, a model is both an abstract and an essential specification; it is a type as well as an implementation or an instance of that model. The separation between a type and an instance (irrespective of the fundamental theoretical difference) does not practically exist, but it is distinguished here. A DEMO model constructed in its default state during development
7. Demo Processor Design and Construction

Model instances can also be linked together, aggregated or destroyed (section 3.2.9, 3.2.10 and 3.2.11) at any time during simulation or production. These aggregated models are a single instantiation of an aggregated type. These self-modifying capabilities at runtime deliver optimal flexibility and evolution capabilities.

7.1.4. DEMO Processor Support for DEMO Aspect Models

The relation between the DEMO processor and the four DEMO aspect models demands a detailed investigation to verify that anything that is in the four DEMO aspect models is represented by the DEMO processor, and that anything that is in the DEMO processor is represented by the four DEMO aspect models. This is a result of cardinality laws 1, 2 and 3 (sections 2.4.4, 2.4.7 and 2.4.8).

The construction model is composed of the interaction model (IAM) and the interstriction model (IM). The interaction model, expressed in the ATD, is completely specified by the $\Psi$ axioms and completely supported by the DEMO processor.

The process model, represented by the process structure diagram (PSD), is a specification of the state space and transition space of all transactions in the c-world. The process model is completely defined and automatically constructed (i.e. calculated) from the ATD. The process model specifies which executions (request, promise, decline, quit, state, accept, reject, stop) any actor is allowed to take at any time, whilst acting in response to other actors’ executions. As the ATD is completely supported by the DEMO processor, the derived process model is similarly, completely and in a detailed manner, supported by the DEMO processor.

The action model controls the process model state transitions, represented in the process model as additional business rules; the ‘free choice’ of an actor, defined by the process model, is optionally bounded. A choice has to be made whether the business rule is enforced or just advised, leaving the option to the actor to deviate. For each possible state transition of each existing transaction, an action model business rule may be specified. A business rule may be dependent upon all kinds of information, originating from either inside the model (via model inspection) or outside of the model. Any information originating from outside of the model, even if it is essential from an ontological point of
view, must be abstracted through an abstraction layer. The action model processor and the abstraction layers deliver full support for the DEMO action model.

The interstriction model is superimposed upon the action model because it additionally bounds the free choice of an actor, and it is executed in the On-event clause (section ). These bonds are calculated by model inspection of transaction types, which are composed of a coordination bank and a production bank and include wait conditions, control of sequential execution, etc. The production bank (p-bank), or the bank contents table (BCT), is a specification for all transactions of object classes, fact types and result types belonging to a specific transaction. Each transaction results in a p-bank that can either be related to a specific transaction or to some external information bank. The BCT contains information necessary to calculate any influence bonds. The issue of calculation within or outside the DEMO processor (model inspection) leads to the abstraction of calculations to external infological systems.

The coordination bank, the second bank required to calculate the interstriction model influence bonds, is composed of all transaction state information for each transaction. The coordination bank is directly accessible and results are directly calculated using if-then-else constructs. The action model processor also directly executes the interstriction model.

The state model specifies classes, n-ary result types and n-ary fact types, together with occurrence and unicity laws over populations, represented in WOSL (Appendix IV). The state model delivers very important insight and identification of derived fact types. It is possible to reconstruct the state model completely from the total population of all possible DMOL states; a single DMOL model instance is not sufficient. However it is possible to directly report the occurrence of instances of result types and to calculate fact types of a state model by the DEMO processor, using ‘model inspectability’. The framework used to construct the DEMO processor allows unary fact types, represented as properties of instances of a class, to be declared and implemented as attachable properties (Appendix III). Optionally, binary and higher n-ary fact types may be calculated and reported to an outside infological system. The support of the DEMO processor for the state model is complete, and it is mandatory for the identification of result types and fact types. The calculation of n-ary fact types are a way to monitor execution from outside of the model; calculated fact types report essential events for outside process monitoring such as management information systems (MIS). For the reconstruction of the state model from DMOL models, the total population of a DMOL model for all possible states is required. The action model processor and the abstraction layers deliver full support for the state model.

To summarize, the DEMO processor is constructed on top of the Ψ processor by integration of the action model processor and various abstraction layers (sections 7.4.1 and 7.4.2). All four DEMO aspect models are required to construct a complete DMOL model instance; anything that is in the four DEMO aspect models is represented by the DMOL DEMO
processor model. Assuming that the complete population of a DMOL model in any possible state is known, it is possible reconstruct the original four DEMO aspect models completely. This bi-directional mapping is loss-less.

<table>
<thead>
<tr>
<th>DEMO aspect model</th>
<th>DEMO model diagram</th>
<th>Description</th>
<th>DEMO processor support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction result table (TRT)</td>
<td>Transaction result table (TRT)</td>
<td>A table stating the transaction types</td>
<td>Implemented by the action model processor / State model processor that calculates instances</td>
</tr>
<tr>
<td>Construction model (CM)</td>
<td>Actor transaction model (ATD)</td>
<td>Specifies actors, productions and communication worlds</td>
<td>Complete support from ( \Psi ) processor</td>
</tr>
<tr>
<td>Interaction model (IAM)</td>
<td>Actor transaction model (ATD)</td>
<td>Specifies dependencies</td>
<td>Complete support from Action Model / State model processor.</td>
</tr>
<tr>
<td>Process model (PM)</td>
<td>Process state diagram (PSD)</td>
<td>Specifies C-world state and state transition spaces</td>
<td>Complete support from ( \Psi ) processor</td>
</tr>
<tr>
<td>Action model (AM)</td>
<td>Process state diagram (PSD)</td>
<td>Specifies conditionally allowed C-world and state transitions at atomic level</td>
<td>Implemented by the action model processor</td>
</tr>
<tr>
<td>Interstriction model (ISM)</td>
<td>Actor bank table (ABT) plus bank contents table (BCT) plus coordination bank</td>
<td>Controls influence bonds as business rules for actor roles, depending on any transaction results and states</td>
<td>Implemented by the action model processor</td>
</tr>
</tbody>
</table>
7. Demo Processor Design and Construction

<table>
<thead>
<tr>
<th>State model (SM)</th>
<th>State model (SM)</th>
<th>Specifies classes, fact types and result types</th>
<th>Implemented by the action model processor / State model processor</th>
</tr>
</thead>
</table>

Figure 7.1. DEMO processor support overview.

Remarks

i.) The construction model is the point of departure for model development.

ii.) Interaction model specifies dependencies between transactions, calculated by the action model / state model processor.

iii.) The process model is automatically constructed / calculated from the IAM / ATD.

iv.) The action model specifies business rules to be applied to specific transaction acts of the process model. It is partly supported directly by the action model processor and in part indirectly using abstraction layers to external systems.

v.) The interstriction model is partly supported directly by the action model processor and partly indirectly using abstraction layers to external systems.

vi.) The state model is completely supported for calculations of instances of result types and fact types. This is required for any MIS type systems and economic dashboards.

vii.) Transaction Result Table is supported by the action model / state model processor that calculates instances by model inspection and stores these instances in external databank systems.

7.2. DEMO Processor Capabilities and Design Requirements

There are two specific requirements for the DEMO processor. The first is a software tool for incremental DEMO model construction, simulation and validation. The second area of interest is in the DEMO processor, executing unlimitedly large DEMO models as the core of enterprise information systems in operation. Based on these two areas of interest design specifications are described. Further insight and understanding may lead to somewhat different implementations. The nature of this section is a pragmatic description of design options and choices and the motivations thereof, as well as directions for further research. This description should be sufficient for qualified software engineers to develop the enhancements on top of the $\Psi$ processor and to resolve any remaining design questions.
7. Demo Processor Design and Construction

7.2.1. Incremental Construction of DEMO Models

The DEMO processor should support the construction of all DEMO models, of unlimited size, starting with the elementary transaction given in figure 3.4 (section 3), whilst guaranteeing correctness. Repetitive application of the production rules (section 3.2.14) allows incremental building of any formally correct DEMO model. The model edit operations are the ‘link transaction’ using the LinkPfact() function, and the ‘delete transaction’ using the DelPfact() function, as well as functions to initialize attributes. A LinkPfact() operation may link either a new individual actor or a complete, alternate DEMO model. The minimum number of model edit operations required to create a model is equal to the number of transactions in the model. This prevents any combinatorial explosion of the number of operations to be performed.

7.2.2. Execution, Simulation and Validation of the DEMO Model

DEMO delivers four consistent and coherent aspect models (section 2.4), each providing a different view of the same ontological DEMO model. The four aspect models are typically shown in graphical representation – a formal graphical language – suitable for shared reasoning and model validation, but unsuitable for processing by an automaton. The NLDMOL (Natural Language DMOL - section 3.3.4) representation of a model in operation delivers specifications to validate the four aspect models. The validation of the aspect models and simulated dynamic behavior may lead either to acceptance of the current DEMO model or to an action of further editing.

Figure 7.2 shows the DEMO processor in execution, by which action the current DEMO model is simulated for model validation. This simulation renders state and state transitions of transactions as well as transaction results. The DEMO model is the complete model in some internal representation.

The DMOL XML model (a file or set of files) can be rendered from this DEMO model, and the DEMO model can be reconstructed by parsing from the ontological DMOL model (section 2.3). Some constructs in DMOL are a declaration and initialization of attributes of a software object from a component library.
7. Demo Processor Design and Construction

7.2.3. Scalability and Enterprise Model Nesting

The size of the DEMO models under investigation and the scalability of the DEMO processor should be unlimited. The ontological DEMO model for a single enterprise is by its very nature of a limited size, as a result of the distinction axiom. In detailed implementation enterprises operate almost always within chains of enterprises, working in a nested or parallel way, as is clear from the composition axiom. Recursive and fine-grained DEMO modeling delivers detailed DEMO models, specifying the operations of each enterprise by including infological and datalogical transactions. These enterprises can be linked by transactions; one enterprise may request a production from another in a recursively nested structure and construct transactions at runtime. The overall structure is that of a single aggregated enterprise, consisting of recursively nested enterprises and resulting in a nested and formally correct DEMO model. Dynamic linking refers to the possibility of creating and executing a new transaction automatically – without human intervention – at runtime and then implementing it in an action model rule. This process takes place when it becomes evident, calculated by a business rule that there is a necessity for a new specific production and transaction to be linked in.

Figure 7.2 Overview of the DEMO processor capabilities.

![Diagram of DEMO processor design and construction](image-url)
7. Demo Processor Design and Construction

7.2.4. DEMO Processor Capabilities of a Production Version

For use in a production environment or in the simulation of large models, a set of additional DEMO processor capabilities is required to achieve perfect expressiveness in order to model real world situations. The list specified is not exhaustive. Application of good design principles, rigorous analysis to guarantee correctness and the elimination of anomalies are mandatory.

Event and state Information Logging
During execution the model can be extended with additional information, such as detailed information of time and conditions of state transitions or even non-ontological data. An option is to store data in additional attributes related to each cfact. In Appendix I, this event logging data is being used for post execution analysis and for verification of correctness.

Model Inspectability and Addressing Schemes
The inspectability of transactions by the action model processor demands a mechanism by which to address a specific transaction instance and its optional properties. The most elementary mechanism uses the transaction identifier; however in the scenario in which more than one identical instance occurs in a model, this is insufficient. A relative addressing scheme should be supported, including the option to specify a transaction using an ordinal number of a child transaction. A relative path-addressing scheme, as described in section 7.4.5, should be adequate.

Dynamic Transaction Linking and Destruction
Dynamic transaction linking refers to the possibility of creating and executing a new transaction between two actors automatically – without a human manual edit operation – at runtime, controlled by an action model rule. The LinkPfact() and DelPfact() functions of section 3.2.10 and 3.2.11 are used to put this into effect.

Application-independence and minimized Expressiveness
It would be completely impractical for a DEMO processor software artifact not to be fully independent of the DEMO model; it is infeasible to reprogram a new DEMO processor for each new specific DEMO model. This argument is sustained with regard to model inspection, where model inspection encompasses access to the so-called transaction result tables (TRT). It is also a good engineering decision to keep the DEMO processor, especially in its first incarnations, as simple as possible and to ‘abstract’ out any complicated constructions.

Important Issue for Linked and Destroyed Transactions
The destruction of a transaction instance during execution, i.e. while the transaction is in any state, can be done at any time without compromising the transaction states of other
transactions. However, the linking of a new transaction is only allowed while the state of
the parent transaction is S0, S1, S2, S3 (section 4.5.8), during the negotiation phase.

**Multiple Model Tree Support**
Most DEMO models show multiple independent trees that are interrelated by conditions,
results and synchronization. The addressing of actors and transactions by action model rules
demands a relative path, and the DEMO processor handles a single tree structure for this
purpose. A convenient way of handling this is to construct a global root transaction where
all local transaction trees are linked. The global transaction is disabled and cannot be
executed, but it serves as the pivot by which actors and transactions are addressed.

**Transaction Enabling and Disabling**
It is mandatory to control transaction execution by setting and resetting a BOOL type
attribute ‘Enabled’, in order to permit the conditional execution of a transaction (section
4.5). A disabled transaction is ‘ignored’ by the initiator, i.e. it does not participate in the
state transition axioms of section 3.3. A transaction can be enabled at any time but the same
cautions as formulated in section 7.3.4, about linking and destroying transactions, must be
taken. The same note as for 7.3.4 applies here.

**Transaction Synchronization and Wait Conditions**
It is required for transaction synchronization, as specified by the interstriction model, to
delay transaction processing using a BOOL type attribute ‘Wait’. If the Wait attribute is set,
the further processing of the transaction is suspended until the attribute is reset.

**Sequential Transaction Execution**
Assuming a model in which a ROOT transaction requires multiple child transactions to be
executed and where a sequence in execution exists, the production of child pfact A is
required for another child pfact B. Conditions like these are specified in the interstriction
model. This situation can be controlled using the Wait attribute. In the OnPromised action
model rule of pfact B, the Wait for this pfact is set. In the OnAccepted action model rule of
pfact A, the Wait attribute of pfact B is reset and the transaction execution of pfact B may
continue.

**DEMO Processor Abstraction Layers**
The DEMO models under execution or simulation communicate with the outside world, or
deal with infological or datalogical data from information systems in the outside world. Yet
any outside world knowledge, processing or implementation-specific issues should be kept
out of this model environment. To achieve this, two abstraction layers are designed.
Abstraction of the Actor Layer and Model Compliance
During simulation or production the communication with the actors should be abstracted. The type of communication with the actor (email, GUI, or any asynchronous interface etc.) should be free. Figure 7.3 shows the DEMO processor in communication with actors; on the left is displayed the technical implementation, and on the right is the formal representation of section 7.1, figure 7.4. In orchestration mode, the DEMO processor controls the coordination between the actors via the interface layer; this capability is already implemented in the Ψ processor. An actor cannot perform any coordination act (a request, a promise, etc.) directly upon another actor, but communicates exclusively through this interface layer with the DEMO processor. The DEMO processor then transfers the appropriate resulting coordination act to the other actor(s), while changing the state space of the DEMO model. The DEMO processor assures that all coordination acts are performed according to the transaction pattern specifications (section 4.5).

Model Compliance
Model compliance is defined as a validation that the actual execution of process steps of the enterprise complies completely with the model [Op ‘t Land, 2008]. Illegal state transitions that violate the transaction axiom are impossible, so model compliance is enforced [Guerreiro et al., 2010, 2011, 2012].

7.3. Action Model Business Rule Calculations and Abstractions

The action model specifies business rules that have to be obeyed in addition to the transaction pattern rules. Both the action model and the interstriction model contain instructions, a set $R$ of rules, to be executed when a certain state transition occurs (figure 7.5). The rules have to be calculated depending on instance specific data, kept in so-called databanks. DEMO action model rules specify these calculations for practical shared reasoning.

However, practical problems occur when this type of calculation is executed by the DEMO processor:

i.) the existence of complex and expensive legacy systems;

ii.) the elimination of data redundancy;

iii.) the possible complexity of calculations, which may demand re-implementation of legacy systems;

iv.) adhering to the good design principle of having as much as possible well-defined abstraction layers;

v.) adhering to the good design principle that the DEMO processor should be as ‘thin’ as possible; it must be perfectly concise.
These problem criteria support the decision that any information specific calculations should be done outside the DEMO processor, where outside is defined as external infological systems. Infological systems do relevant calculations, while the implementation of these calculations is not of interest. There is a functional interface specification of the infological system. Infological systems return a BOOL type to be used for conditional execution.

The action model is executed by the action model processor, executing as an event handler. At a specific transaction event the associated action model rule is executed. From the above problem criteria, it follows that the action model is as ‘thin’ as possible and any additional calculations should be abstracted out to infological systems.

A specific rule $R$ either replaces or enforces the actor’s decision, or ‘advises’ the actor to decide in a specified way while the actor may deviate. There is model design freedom. There is room for debate as to whether the processor should enforce a business rule, or whether the (human) actor should be able to decide to the contrary anyway and to thereby take responsibility for that decision. In the example shown below (figure 7.5), the actor decision to decline or to promise is taken by the execution of $R$.

![Diagram](image)

**Figure 7.3.** DEMO processor with abstraction layers for actors and infological/ datalogical systems.

Figure 7.3 shows calculations that are partly infological (or datalogical – the distinction is not of any interest here). Hence an infological abstraction layer is needed, as is a generic action model language. This is achieved by calling infological systems and passing a specification of the call and the set of parameters. The infological systems return a BOOL variable for conditional execution of the action model rule by the action model processor.
An example of action model specifications is given in figure 7.4.

```
on requested T01(M) with member(new M) = P
    if age(P) < minimal_age -> decline T01(M)
      ? not -> promise T01(M)
    fi
no

on promised T01(M)
    if membership M applies for reduced fee for year Y -> request
      T03(M,Y)
        ? not request T02(M,Y) with standard_fee(Y)
      fi
no
```

Figure 7.4. Example of ontological Action Model rules. Example from Dietz [2006]

Figure 7.5 shows the calculation of external implementation specific information within the ontological model.

```
OnRequested T01
    if Infologic( Opcode, ParamRef ) == true then
        SetStatus( T01, Decline ) /* Executor declines T01 */
    else
        SetStatus( T01, Promise ) /* Executor promises T01 */
    fi
No

OnPromised T01
    if GetStatus( T05, Accept ) == true then
        SetStatus( T03, Request ) /* Initiator issues requests for transaction T03 */
    else
        SetStatus( T02, Request ) /* Initiator issues request transaction T02 */
    fi
No
```

Figure 7.5. Example abstracted Action Model instructions /* with comments */.

For example the calculation of (age(P) < minimal_age) of figure 7.4 must be executed by some external calculating system. By way of abstraction and by using the instruction set (section 7.4.3), a correctly abstracted representation is shown in fig. 7.5.

The abstracted version does not handle implementation dependent variable manipulation (for example is (member(new M) = P), and (age(P) < minimal_age) directly. As the DEMO processor handles instances, these variables or identifiers can also be passed as
parameters. Some of these parameters are instances of fact types which can be reported to or requested by the action model processor from the infological system. The parameter T01 identifies that the transaction is not required; the action model rules are attached to a specific transaction.

**DEMO processor tuple for action model rules**
The action model is composed of a set of rules R and a set of relations between cfacts and rules CR, as specified in section 3.1.2. For each cfact event, a rule from the action model can be related. The \( \Psi \) processor tuple, tuple 2, \(<Pf, Pcom, Cf, CP, Act, Executor, Attribs, Attr>\), is now extended to the DEMO processor tuple, tuple 1, \(<Pf, Pcom, Cf, CP, Act, Executor, R, CR, Attribs, Attr>\) of section 3.2.2.

### 7.3.1. The Action Model and State Model Processor Instruction Set
The action model processor should execute exclusively DEMO based constructs and use a single construct for communication with the outside world. A pragmatic instruction set for the action model processor is proposed as follows.
The State Model processor calculates fact type instances. Both processor capabilities are implemented in the same software engine.

\[ \text{BOOL infologic( Opcode, ParamRef )} \]
A call to some external infological system, excluding any other ways for I/O. \text{Opcode} identifies the function to be called. \text{ParamRef} is a reference to a specific parameter or parameter structure. The function is being used to request information or to report information such as result type and fact type instances and, in general, transaction bank items. The return type BOOL allows conditional execution. It is mandatory that (i) calling of this function returns in real time, and (ii) execution of this function is not likely to return failure of execution due to external problems. If this might be the case, a separate transaction should be constructed for a ‘stateful’ communication. This transaction executes an asynchronous function where timeouts do not compromise system integrity or result in hanging.

\[ \text{BOOL GetStatus( TransactionIdentifier, StateIdentifier )} \]
This function returns the state of a transaction as a confirmation of the value of parameter StateIdentifier. The parameter StateIdentifier in the example must be one of the set of 12 allowed transaction states (S00, ..., S11) as specified in section 3.3.11. The parameter TransactionIdentifier refers to a specific transaction through an addressing scheme.
Void SetStatus( TransactionIdentifier, StateIdentifier )
This function assigns a new transaction status. The new transaction status must comply
with the state transition specifications.

**if-then-else control structures**
This construct should be adequate. Alternative control structures are undesirable and this
even includes while-wend or for-next. Execution of an event is atomic; there should be no
polling or waiting loops for model inspection. Any interfacing with external infological
systems should also be abstracted.

BOOL LinkPfact( b, q, t )
This function links or adds a DEMO tree q – practically, a link to a copy of a component in
a type library – to Initiator b, as specified in section 3.1. Parameter t assigns a transaction
identifier. The LinkPfact() function allows incremental contraction of transactions or
DEMO components at runtime. The return value is true if the linking is performed as
intended. The return value is false if the operation fails, if p and/or q are not found.

BOOL Copy( a, q )
Copy() copies pfact q, including of its attached executors and their child objects, to
executor a and constructs a transaction. The Copy() function generates a new instance of q.

BOOL DelPfact( p )
DelPfact( p ) detaches and destroys pfact p and any underlying transactions, as specified
in section 3.1.6.3. The DelPfact() function allows incremental destruction of DEMO
components at runtime. The return value reports the result, executed correctly (true) or
incorrectly (false). A convenient implementation would be parameter passing using the
transaction identifier, since this value is locally defined within the On-clause.

BOOL Exist( p )
This function checks if an object with identifier or IdentifierPath p exists in the model. At
any time during execution, it is not known in advance whether a specific transaction or
component exists. This function is required for conditional linking of DEMO components.
If the component exists already it does not need to be linked in again.

**Operators AND, OR, NOT**
These operators are required to handle complex conditions.
BOOL Save( parameters )
The Save function renders a DMOL representation or logs data to persistent memory. The
parameters specify also which type of information is rendered and to which location in a
file system or database.

Timeout capabilities
While transactions are formally asynchronous without timeouts, these transaction timeouts
are often mandatory in a production environment. A programmable timeout function should
generate an event, and that event should handle the situation in an appropriate way.
Recovery from a timeout condition must be implemented in action model rules.

Relative addressing scheme
For model inspection, an addressing scheme is required. A suitable relative addressing
scheme is:

\[
\text{Variable} = |\text{Tn}.\text{Tp}.|\text{MemberFunction()}
\]

For example:

\[
\text{StatX} = \text{T01.T10.T21.GetStatus()}
\]

In the current transaction where this action processor code is executed, the following
algorithm is followed:

Is T01 the identifier of the current transaction?
\begin{enumerate}
\item[i.] If not, then recursively the parent transaction is checked until T01 has been found.
\item[ii.] If T01 has been found, its child transactions must be checked to find T10.
\item[iii.] If T10 has been found, its child transactions must be checked to find T21.
\item[iv.] If T21 has been found, the value of the request must be assigned to \text{Variable}.
\end{enumerate}

The existence of the relative path can be checked prior using the \text{Exist() } function:

\begin{verbatim}
if Exists( T01.T10.T21 ) then
else
  /* Transaction does not exist */
fi
\end{verbatim}
There is no claim of completeness or correctness of this instruction set. Empirical validation is needed.

### 7.3.2. Action Model and State Model Examples

Figure 7.8 gives an example of the conditional reporting of the existence of a binary result type. It should be noted that this is not an action model business rule, but the implementation of this type functionality should be included in the action model processor. This report capability should be implemented also in the NLMOL rendering.

```plaintext
OnAccepted T01
  if GetStatus( T05, Accept ) == true then
    Infologic( Opcode, "Binary Result fact for T01 and T05 exists."
    /* Both transactions have been accepted and the binary result fact has been created */
  fi
No

Figure 7.6. Example showing instance of a binary Result type being reported.
```

In figure 7.6 there is a condition that should be met – transaction T03 must have been completed, accepted – in order to promise transaction T01. The result of transaction T03, Accept == true, is required for the production of transaction T01.

```plaintext
OnRequested T01
  if GetStatus( T03, Accept ) == true then
      SetStatus( T01, Promise ) /* Executor of T01 promises T01 */
  else
      SetStatus( T01, Decline ) /* Executor T01 declines T01 */
  fi
No

Figure 7.7. Example showing interstriction condition implementation for a unary Result type.
```

Figure 7.8 gives a situation comparable to that of figure 7.7, but in this case a binary result type instance, T03 and T05, is required.

```plaintext
OnRequested T01
  if GetStatus( T03, Accept ) == true then
    if GetStatus( T05, Accept ) == true then
      SetStatus( T01, Promise ) /* Executor of T01 promises T01 */
    else
      SetStatus( T01, Decline ) /* Executor of T01 declines T01 */
    fi
  else
    SetStatus( T01, Decline ) /* Executor of T01 declines T01 */
  fi
```
else
    SetStatus( T01, Decline )    /* Executor T01 declines T01 */
fi

Figure 7.8. Example showing interstriction condition implementation for a binary Result type.

Figure 7.9 gives an example of the destruction of a transaction T03, apparently after a failure to execute (a decline). This is followed by a retry, by constructing a new transaction with another DEMO component from a library, and setting the request for the new transaction T03.

OnDeclined T03
    Destroy( T03 )    /* destroy transaction T03
    LinkPfact( b,q,T03 )    /* link a new T03 instance from
    SetStatus( T03, Request )    /* set a Request, retrying the
                               transaction */
fi

Figure 7.9 Example showing destruction and linking of a transaction.

External Process Control and MIS systems
In the first action model example of figure 7.8, the report of a result type is calculated ("Binary Result fact for T01 and T05 exists"). This can be part of an external process audit log, MIS, or it can be used as a trigger for a real-time event-driven process monitoring system. Process monitoring is hence built in to the model.

Extended Capabilities for Evolving Components
The principle of evolving components is based on the need to change the implementation of a component randomly and unpredictably at runtime. The black box specifications remain identical, while the white box specifications are upgraded and replaced by a newer and better version. This capability demands the following implementation:

At runtime, any component checks in the library if a newer version exists with a higher version number.
If this component exists, it is linked in using the link function of section 3.2.10.
The newer linked in component checks the version numbers, and it may or may not decide to upgrade the existing DEMO component.
The new DEMO component will read all state and other information from the original DEMO component and initialize the new DEMO component accordingly.
After initialization, the original DEMO component is destroyed and replaced by the new DEMO component. For this capability full control over the DEMO component is required; reading, and assigning status data (GetStatus() and SetStatus()) and arithmetic functions to read, copy and manipulate integer, real and string variables.

7.4. Remaining Design Topics

The following topics can be resolved in subsequent design cycles following the Design Science Research paradigm, when more case studies have been done.

The Extended Attributes

Extended attributes and functions such as Enabled, and Infologic() parameters (Cfact) and Self-Initiating (Actor), have to be added to the model. This is easily implemented with additional XML branches.

Another category of attributes originates from the fact that a model instance is executed (section 2.4.4). There are usually many model instances that populate the class of a model. Each of these model instances must be identified from other instances using some unique identifier.

In addition the interfaces to the "DEMO external databanks" and the "infologic" information systems require attributes at instance level for actors, cfacts and pfacts.

Deadlock Conditions

In the Ψ processor, internal deadlock conditions are not possible because dependencies and interstriction conditions do not exist and a mutual wait is impossible. In the real world, deadlock conditions where two transactions wait for each other, as expressed by an interstriction rule, are a common phenomenon. It should be possible to model these deadlock conditions too. It is possible to recover from such a deadlock condition during operation using the transaction cancel mechanisms (section 5.6). The automatic detection of potential deadlock conditions in a DEMO model is an important issue, outside the scope of the present research, yet a pertinent subject of further research.

Completeness of Expression of the Action Model Processor

The action model processor should be able to express and to compute any condition or calculation that may follow from the action model rules and state model constructs, perfectly comprehensive. Yet the expressiveness of the action model processor should be as 'thin' as possible, perfectly concise. The current design of the action model processor is terse, it shows the most important design topics, but is not complete. More examples and cases for the action model processor are needed. This requires further research for final proof of completeness.
Maintaining Minimized Expressiveness
If minimized expressiveness is not guaranteed then application specific elements would be implemented in the DEMO processor and increase its overall complexity. These elements can never be removed later without a potential loss of backward compatibility for existing applications.

Various Implementation Issues
In Model driven engineering there is (usually) a series of transformations from high level abstractions to low level abstractions. In the first implementation of the $\Psi$ processor, used for this research, a proprietary object-oriented model-driven framework has been used. This framework already possessed many capabilities needed by the $\Psi$ processor. One of the special capabilities is that this framework is fully interpreted and instance-driven; the type declaration is identical to the model instance; i.e. a model instance is its own type declaration. There is no translation to lower abstractions. High level abstractions, represented by software objects are constructed by aggregation of software objects called “node”. Nodes may have scalar attributes (real, integer, string). Nodes are related using parent-child links. Nodes have member functions that can be called (recursively) and executed. Each node has a set of events that may trigger member functions. Aggregated software objects can serialize and reconstruct themselves to and from a XML (like) language. Aggregated software objects can modify themselves at run time or link to other software objects, or destroy parts of themselves. The expressiveness of this framework is sufficient and even quite convenient to construct these type of systems. The interpretation way of working makes it very easy for development but not suitable for production due to its slow execution speed.

For the professional production environment of the DEMO processor a version in compiled C# is being developed. This implementation needed much more time for implementation but provides a much better execution speed. Also in this version, there is no runtime translation to lower level abstractions.

7.5. Conclusions and Results
The DEMO processor is an extension of the $\Psi$ processor, with enhanced capabilities to implement business rules specified by the action model rules and constraints specified by the state model. The DEMO processor supports all four DEMO aspect models and is a professional software environment for modeling, simulation and enterprise compliance in full production. The required professional capabilities and global design solutions have been formulated, but need validation and further engineering cycles. Some design issues demand further detailed research cycles, especially the action model processor and state model processor. With good confidence can be claimed that the feasibility of the DEMO processor that fulfills its requirements has been shown.
PART V

Conclusions and Future Research
8. Conclusions and Future Research

8.1. Research Questions Revisited

In this thesis, the engineering of enterprise information systems (EISs) has been investigated. The motivation for this research, expressed in the research objectives (section 2.1), is the fact that information systems fail too often in three ways; there are three problem symptoms. The first is that they do not support the operation of an enterprise in an adequate way. There is a functional mismatch problem. The second problem symptom is the failure to engineer information systems, which involves programming, in a controllable, cost-effective and high quality way; projects exceed allocated resources or fail to deliver. The third problem symptom is that information systems have to evolve to support the normal evolution of the enterprise, the agile enterprise, over time, but exhibit increasing costs for software maintenance and adaptations.

It has been argued that these symptoms of problems are largely caused by the state of the art’s inability to deliver high quality functional requirements and constructional specifications for information systems. With high quality functional specifications, we address the first problem symptom of functional mismatch. With high quality constructional specifications, we address the second problem of uncontrollable and cost-effective information system engineering. The quality of the constructional specifications refers on one side to the C4-ness qualities, the completeness, consistence, comprehensiveness and coherence qualities of the specifications. With C4-ness quality specifications, software engineers are usually able to construct information systems that comply well with the constructional specifications. Examples are the complex Internet, GPS and cell telephone systems. The costs of these IT systems may be high but are usually manageable and controllable.

On the other side, there is the problem that, even if we have C4-ness quality construction specifications, we may end up, after programming, with an information system that exhibits a serious functional mismatch. The construction specifications already exhibit the functional mismatch problem.

---

92 Managerial, political and other non-engineering problems that may cause project problems are not investigated in this research.

93 Functional requirements specify the function of the information system, its behavior when interacting with the enterprise. This is a so-called black box specification.

94 Constructional specifications specify how the information system is constructed, i.e. its parts and the interoperation between the parts and the ‘outside’ environment.
It is hypothesized that this inability to deliver high quality constructional specifications without functional mismatch for enterprise information systems, is due to a lack of scientific and engineering foundations of the methods applied so far. Because, in other more mature engineering disciplines, the scientific and mathematical foundations are well established, as we observe, but apparently not (yet) for software engineering. The many state of the art methodologies apparently lack scientific and formal foundations.

It was recognized that the first two problem symptoms could be addressed by the use of a domain ontology. Any domain ontology has two aspects or views, a philosophical aspect and a formal aspect. The philosophical aspect addresses the question if and to what extend the ontology provides us with an appropriate and truthful representation of that part of reality of interest. The formal aspect addresses the question if any representation of our part of reality is a high quality C4-ness representation (or not). The most important philosophical qualities, expressed by the appropriateness and truthfulness quality criteria, enables to address the functional mismatch problem symptom. The C4-ness quality representation can provide us with the foundation to obtain high quality constructional specifications.

The advantages to use a domain ontology as the foundation to address the first two problem symptoms made clear that we must first understand enterprises well. Just as good as other domains are understood in mature and successful engineering disciplines (aircraft engineering, electronics, civil engineering etc). We need an ontology for enterprises as our foundation. The theory of enterprise ontology (section 3.6) has been assessed, found to have a good appropriateness and provides our domain ontology.

An important observation is that enterprises are systems that are purposefully constructed to fulfill some useful goal. Enterprises should implement their strategy, which is a functional requirement. This consideration led to the application of the generic systems development process (GSDP, section 1.2.6) as an engineering foundation. The model driven engineering (MDE) [OMG, The Object Management Group, 2001], (section 1.2.7) approach delivers the way to construct a software engine that executes models expressed in an appropriate high level modeling language. The Design Science Research Paradigm (section 1.2.8) provides the guiding principles of engineering cycles and incremental improvements.

The fact that an ontology provides the C4-ness quality model specifications we need so urgently, led to the use of ontological models as foundation of information system specifications, in stead of lower level specifications in generic programming languages. The application of Guizzardi’s framework for ontologies, ontological languages and structural conceptual models provides very strong benefits.

95 If the representation lacks C4-ness quality then the ontology is not an ontology in the formal sense. That ontology is then (almost) worthless for our purpose. (section 3.4.8).

96 The appropriateness for EISs still has to be shown based on a sufficient number of empirical cases.
8. Conclusions and Further Research

Three Research Questions

The first research question regards the problem of how to construct high quality information systems, in such a way that the three problem symptoms are addressed. The answer is the GSDP-MDE research question. The GSDP-MDE methodology is an application domain independent methodology to construct information systems directly from high-level ontological models using a domain ontology. This engineering artifact is a design science method.

The second research question regards the application of the GSDP-MDE methodology for enterprise information systems using the theory of enterprise ontology for enterprise information systems. The first part of the answer is the design, the specifications of the \( \Psi \) processor, the core of an enterprise information system that is directly derived from the \( \Psi \) theory. The formal specifications of the \( \Psi \) processor are an engineering artifact, a design cycle. The second part of the answer to the second research question is the resulting software implementation, a design science instantiation. This part includes extensive simulations, verification and validation of the \( \Psi \) processor.

The third question addresses the engineering of a software application development suite for professional applications that is based on the \( \Psi \) processor, the DEMO processor. The DEMO processor supports all four DEMO aspect models. This design is an engineering artifact, a design cycle. It should be noted that this is the first design cycle, a version with limited capabilities exists and design cycles for further improvement are most likely needed. There is no claim for completeness of the design.

8.2. The GSDP-MDE Methodology Research Question

As discussed before, the foundations for the first research question are the use of a domain ontology, the GSDP engineering process, the model driven engineering approach and the Guizzardi framework. The first research question that addresses the questions how to construct software systems from systems ontologies for any domain:

*How to construct high quality IT system models, a high quality modeling language and an executing software engine, suited for any domain?*

The scientific and engineering foundations required to answer this research question are:

\[ i. \] The first foundation is the Generic Systems Development Process (GSDP) (section 1.2.6). The GSDP is a generic model for developing any kind of system that supports
another system. In this case a software system that supports the operation of an enterprise (a social system).

ii.) The second foundation is an ontology to capture our domain of interest. An ontology provides a formal representation of our domain of interest which provides the C4-ness quality specifications (section 6.4) we need. An ontology should also capture the phenomena in reality in a truthful and appropriate way, fit for addressing the functional mismatch problem.

iii.) The third foundation is the Guizzardi framework (section 1.2.2) for ontologies and structured conceptual models. Guizzardi postulates a cardinality law for static ontologies. If this law is applied then the metamodel of an ontological language L can be constructed to express conceptual models with many benefits and lack of anomalies.

iv.) The fourth foundation for the engineering of software systems is model driven engineering (MDE) (section 1.2.7). In MDE, a model of a system is congruent to the input, the 'source code' in a specific formal language. A dedicated software engine executes models expressed in that language directly, either by interpretation or translation to executable code.

v.) The fifth foundation is provided by three scientific theories, the $\Phi$, $\tau$, and $\Psi$ theories (sections 1.2.3, 1.2.4 and 1.2.5) that provide the scientific foundations of ontology and especially the theory of enterprise ontology.

To answer this research question the following steps have been made:

i.) The Guizzardi framework and the cardinality law for static ontologies has been extended for dynamic ontologies, which results in two additional cardinality laws that apply to dynamic ontologies. The before-mentioned language benefits are also obtained for the construction of the modeling language $L$ for dynamic ontologies, such as enterprise ontology.

ii.) The Guizzardi framework has been further extended using the GSDP methodology and the model driven engineering approach, which results in the GSDP-MDE methodology for software engineering. It integrates a modeling language and an executing software engine.

iii.) It has been shown, and as expressed by the three cardinality laws, that the domain ontology, the ontology of the metamodel of language $L$ and the ontology of the executing software engine, are all three isomorphic ontologies.

iv.) It has been shown that a domain ontology with a high quality appropriateness and truthfulness, addresses the functional mismatch problems.

v.) It has been shown that rigorous application of the three cardinality laws and a truthful domain ontology with a high degree of appropriateness, delivers a modeling language with the valuable language benefits.

vi.) It has been shown that application of the three cardinality laws of the GSDP-MDE methodology provides model specifications that have C4-ness qualities.
The benefits of the modeling language $L$ (section 3.4) include:

1. Elimination of programming and improved software quality; the model is directly only translation of representation is required, no programming – the executable 'source code'.
2. Maximum reuse of programming efforts; the metamodel of the language $L$ and the executing software engine have to be developed only once for a specific domain ontology.
3. Elimination of anomalies; such as construct excess, construct overload, inconsistencies, ambiguities and C4-ness quality.
4. Minimized expressiveness advantage. $L$ has complete ontological expressiveness within and zero expressiveness outside the domain of the ontology. It is impossible to construct models of phenomena that cannot exist in reality.
5. Optimal complexity reduction and zero entropy; due to the fact that there is a 1:1 cardinality between every phenomenon that may exist in reality and a corresponding model.

The developed GSDP-MDE approach enables “model instance driven software engineering”. A model instance, expressed in the modeling language, is the "source code" to be executed and no subsequent software coding for the EIS is needed. Any unique model instance is at the same time its own unique model type specification; i.e. a model instance specifies itself. Any execution progress and resulting state change of the model instance is reflected in a recalculated model instance while maintaining a perfect compliance between the model under execution and the production instance in reality. I.e., a model modifies itself over time. It is possible to construct new model types 'on the fly', at production time, that enable new production instance types, which supports the agile enterprise by evolution of information systems.

This approach differs in several ways from most model driven engineering approaches where an executable (compiled and static) program for a class of instances is generated in a lower generic language. The first difference is that there is one and only one model instance for each corresponding production instance. The second difference is that the model

---

97 The elimination of programming applies strictly to the domain of interest. In practice software systems have many interfaces with other systems, user interfaces etc, that require traditional software programming techniques. In addition, there are so-called MIS systems that do not describe the Actor, Transaction and Production worlds and for these category systems this claim does not apply.

98 The elimination of programming applies strictly to the domain of interest. In practice software systems have many interfaces with other systems, user interfaces etc, that require traditional software programming techniques. In addition, there are so-called MIS systems that do not describe the Actor, Transaction and Production worlds and for these category systems this claim does not apply.
instance is modified each time the state of the production instance is changed. The third difference is that model instances can be subjected to self-modification during production. The fourth difference is that there are no series of model transformations or translations to lower-level languages. The self-modification of models under execution is driven by execution of business rules, that are part of the model itself. The possibility of runtime self-modifying software supports the notion of the agile enterprise.

The consequence of the GSDP-MDE approach and its valuable benefits are however the substantial efforts for any implementation:

i.) To devise a domain ontology for our domain of interest.

ii.) To assess the truthfulness and appropriateness qualities of the ontology.

iii.) To verify the formal C4-ness qualities of the ontology.

iv.) To devise the ontological modeling language.

v.) To design and implement the executing software engine.

vi.) To validate its appropriateness and truthfulness qualities and verify formal qualities using simulations.

The advantage is that these steps have to be performed only once for any ontology. This represents the highest degree of software reuse possible, known so far.

As stated repeatedly earlier, these advantages apply exclusively to EISs. In typical information engineering many tedious traditional programming tasks remain.

8.3. The Ψ Theory Finite State Automaton Research Question

The first research question investigates the use of a systems ontology as the foundation of software engineering. The second one addresses the application of these results to enterprise ontology, in fact the Ψ theory to construct a software engine, a finite state automaton\(^99\) that executes models in a modeling language.

How to construct a finite state automaton, or a set of finite state automata, and a modeling language using the GSDP-MDE methodology, that is compliant to the Ψ theory?

The Ψ theory consists of four axioms and one theorem, concerning the construction and operation of an enterprise. The three cardinality laws investigated in the first research

\(^{99}\) For models composed of transactions there is actually a single automaton for each transaction. The term finite state automata system is applicable too. There is however one and only one software system.
question have been applied to the $\Psi$ theory. The result is a set of axiomatic specifications for software primitives and constructs of the DMOL (DEMO Modeling Language, represented in XML\textsuperscript{100}) language. In section 4 the design of static model specifications has been presented. In section 5 the design of dynamic model specifications has been presented. In section 5 the validation, verification and a quality assessment have been investigated also.

Every phenomenon in reality that is captured by the $\Psi$ theory is the organization of an enterprise. It can be specified by a model expressed in the DMOL (XML) language. Especially the formal correctness and the C4-ness (comprehensiveness, coherence, consistency and conciseness) quality criteria have been investigated. Additional reasoning for the C4-ness claim of DEMO models has been supplied. The elimination of software programming, absence of anomalies, absence of undesired software constructs excess or overload, absence of increasing complexity due to subsequent modifications has been investigated and confirmed.

To support empirical verification of the quality of the formal specifications, the executing software engine, the $\Psi$ processor, has been implemented. With this $\Psi$ processor, $\Psi$ model instances can be constructed, modified and simulated. The models can be read from DMOL file representations, and at any time, a new modified DMOL file representation can be regenerated. DMOL models can be constructed by creation, aggregation or destruction of ‘software component’ DMOL models at run time. Empirical verification, validation and simulations (section 6, Appendix I) delivered good results in the sense that no anomalies have been found. However, more rigid and complete temporal logic simulations are required as a future research topic. It is shown that DMOL models contain a set of transactions. For each transaction one automaton is required. The executing software engine is actually composed of a set of automata.

The formal specifications are an engineering artifact, a design cycle in terms of the Design Science Research paradigm [Hevner, 2004, 2007]; the first part of the answer to this research question. The resulting software implementation is a Design Science Research instantiation.

### 8.4. The Design of the DEMO Processor

The third research question, investigated in section 7, addresses the construction of a software engine with full support for the four DEMO aspect models. The DEMO aspect models specify an enterprise in a more extended way than the $\Psi$ theory. DEMO models include the state model that specifies fact types etc. (section 7.1.3) and the action model that specifies business rules. Therefore the DEMO processor with enhanced and extended capabilities over the $\Psi$ processor and full support of the DEMO aspect models, fulfills the requirements of an enterprise information system. This is expressed by the third research question:

\textsuperscript{100} Extensible Markup Language (XML) is a very flexible and convenient markup language. Developed by the World Wide Web Consortium (W3C) and widely standardized.
How to design the DEMO processor, based on the $\Psi$ processor, that supports all four DEMO aspect models?

The DMOL language (of the $\Psi$ processor) has been extended with primitives and constructs to specify a complete and unique instance of every set of DEMO aspect models. The relations between the primitives and constructs of the DMOL language and the four DEMO aspect models have been investigated and defined. There is no loss of information resulting from the mapping process from DEMO models to DMOL models vice versa, except for the fact that a DMOL model is a model instance and the DEMO aspect models specify the classes of instances and their types. It is shown that the four DEMO aspect models also comply with the three cardinality laws of the first research question and are C4-ness. This supports the claim that the organization of every enterprise in reality corresponds with one and only one correct DEMO model and DMOL model. Vice versa; for every verified DEMO model and DMOL model there may exist an enterprise in reality.

This pragmatic design of the DEMO processor is the result of the first engineering design cycle in the Design Science Research paradigm. There is no claim that this practical design is already complete and entirely correct, it has to be followed by further design cycles with incremental improvements. Future research is needed.

8.5. Elimination of the Research Objective Problems

It is argued as follows that the three research objectives (section 2.1) in theory have been realized.

8.5.1. Elimination of Functional Mismatch Problems

The functional mismatch problem is usually discovered by validation after software implementation and after large costs and resources. After modifications of specifications a new software implementation has to be made. With the DEMO processor (or any other GSDP-MDE system) a functional mismatch problem can be identified immediately after design. Repeated design cycles can be pursued immediately without software implementation. The functional mismatch problem is eliminated because its cause - programming - does not exist anymore.

8.5.2. Elimination of Uncontrollable Software Programming Resources

The apparent lack of C4-ness quality construction specifications is eliminated. Constructional specifications can be directly executed for simulation, validation and production, which eliminates the large resources for programming. The problem is
eliminated exclusively for the domain of interest; outside the domain any existing problems
are not solved.

8.5.3. Elimination of Growing Software Costs, prohibiting Agile Enterprise Support

The uncontrollable growth of software complexity and its increasing costs is also eliminated. Each time an enterprise needs to re-engineer itself, a new model is designed and executed immediately. Enterprises can now be supported by agile information systems.

8.5.4. Expected Benefits of the GSDP-MDE Methodology

We assume that the application domain is “the production of services, that are subject to complex regulations, for demanding customers, who actively participate in the production”. For example, the financial and insurance services for customers is part of this category. Government services are also in this category. Based on our experience and the first professional case (Appendix IV) we estimate:

A typical application consists for approx. 50% of code resources within the domain of enterprise ontology, the first part. The remaining 50%, the second part, involves code for interfacing to legacy systems, production code (pfacts), and various items.

The code production for the first part is eliminated. Instead much more effort will be spend on fine grained DEMO modeling, simulation and validation cycles with the staff (non IT experts) to obtain optimal business processes. This may take up to half of the efforts that would have been spent for coding for the first part. The benefits however are a much better business-IT alignment and virtual elimination of the chance for large rework of programming. The overall reduction in resources for the first part is estimated to be 50%.

For the second part, the production of high quality models of the first part provides an efficient decomposition of programming work for the second part, plus detailed high quality specifications. The remaining resources needed would be half of what it used to be. An important benefit is that the chance that a project runs out of control for programming of this part is virtually eliminated. Programmers freedom is largely eliminated, all applications look and feel very similar. Production of IT systems from DEMO models may become a mass-produced commodity.

Summary of expected benefits:

i.) The overall reduction in resources for a typical project will be around 50%.

ii.) The application will have a substantially better business-IT alignment.

iii.) Production of applications will become more a well-controlled mass-produced commodity, easy to outsource to a specialist company, with good support for the agile enterprise.
A sufficient number of business cases is needed to gain experience and to substantiate this expectation.

8.6. Contributions to Enterprise Information Systems Engineering

In this section we discuss how the research results contribute to solve the three major problems formulated in the research objectives (section 2.1). The DEMO processor supports enterprise information systems engineering, though with a number of issues that need further research, as described in section 8.7. The DEMO processor provides:

i.) Support for domain model development.

The DEMO processor is a DEMO model development and simulation system for model validation, especially the dynamic behavior can be investigated (section 6, Appendix I). The DEMO processor is a model-instance development system; only DEMO model instances specified by the DEMO aspect models can be constructed and simulated. The advantage over traditional model development or ‘paper-based’ model validation is the ability to simulate unlimited large models. Typically a model with more than two transactions is too complex for human calculation. A large and complex aggregated enterprise or a supply chain of enterprises can be represented by a set of enterprise models. DMOL models can be aggregated to represent these large enterprises and the dynamic behavior of the whole production chain can be simulated.

ii.) Support for domain model execution.

The DEMO processor is also a production environment where numerous large aggregated DEMO model instances can be executed. A model instance is executed, controls the production instances (item iv) as defined by Enterprise Dynamic Systems Control [Guerreiro, 2012] and workflow capabilities. The model instance corresponds always precisely to the production instance; i.e. at any moment, any model instance delivers a truthful representation of the related production instance. The DEMO model instances under execution should calculate all production fact type instances specified by the DEMO state model, execute all business rules specified by the DEMO action model and report the calculations to non-EIS information systems such as aggregating production monitoring or value delivery systems (MIS). In this way it is a descriptive information system with ontological completeness; there is complete knowledge of each atomic production and communication fact. A lack of ontological completeness of calculated facts of EISs poses a major problem for any MIS information systems.

iii.) Support for the operation of an enterprise.

The DEMO model-instance driven information systems support the operation of an enterprise (section 1.1.1). The operation axiom states that an enterprise is composed of three worlds; the A-world composed of Actors, the P-world composed of production
acts and facts; the C-world composed of communication acts and facts. The following support is delivered:

i.) For the C-world: a completely specified C-world (intersubjective communication patterns and its coordination, in line with the transaction axiom). The completely calculated workflow support is an alternative for BPMN modeling with significant advantages (item iv).

ii.) For the P-world: a fine-grained and ordered structure of the P-world, composed of structured fine grained atomic p-facts – production items and their relations. The calculated p-facts during execution populate a fine-grained DEMO transaction result table (TRT) which delivers – if properly modeled – a C4-ness specification of the production. The calculated TRT should deliver complete information for any high-level corporate information system.

iii.) For the A-world: a structured and ordered specification of actor roles, with competencies, authorities and responsibilities.

iv.) The interfacing to the DEMO external Data banks and the translation to relevant business concepts deliver specifications that support the non-EIS information systems.

v.) The remaining domains of non-EIS information systems, typically MIS systems and production systems, are not supported. Any information processing that is not within the before described A-, P- and C-worlds is not implemented. Substantial support is delivered by the high quality specifications and the calculation of DEMO State model facts.

vi.) Capabilities for workflow support. Guerreiro [2011, 2012] showed the DEMO processor is a role-based access control system for enterprises; an Enterprise Dynamic System Control (EDSC). The DEMO processor is an alternative approach to state of the art workflow systems, that address the same challenge to control enterprises. The main difference is that the DEMO model enterprise control is completely calculated from DEMO models. With DEMO, there exists no equivalent to ‘workflow modeling’; DEMO models represent enterprises in a much wider scope than e.g. BPMN-based workflow. The harmonization of DEMO and workflow systems is discussed as a future research topic (section 7.4.8).

8.7. General Contributions to Science

The GSDP-MDE methodology is a generic methodology to construct information systems for any domain for which a domain ontology\(^{101}\) exists.

\(^{101}\) We assume an overall absence of any anomalies, for example, there is a limited set of concepts and a deterministic discrete behavior etc. Any anomalies have not been investigated.
Three cardinality laws are postulated in order to obtain ontological models with important advantages. If these laws apply, a computer-readable language composed of a sequence of symbols, for that ontology can be constructed. This offers many benefits, such as the elimination of software construct excess; elimination of anomalies; complete expressiveness within and zero expressiveness outside the domain of the ontology. Within the domain of the ontology, there is zero entropy between a model conceptualization and the corresponding model specifications in that language. The model instance is equivalent to the executable code for a specific production instance. In this way no manual programming, a translation to more elementary primitives, is needed and programming related problems cannot occur.

If these laws apply, in addition to the ontological language a software engine can be constructed to execute models expressed in the ontological language.

In section 6.2, the simulation results of the software engine in Appendix I and the first DEMO processor application in production (section 8.7), additional reasoning is provided for the stance that enterprise ontology is a truthful and C4-ness (coherent, concise, comprehensive and consistent) quality representation of enterprises operating in reality. The ontological appropriateness statement that should provide a 'better' business IT alignment for EISs is so far only founded on existing empirical evidence for enterprise ontology for other applications but needs more empirical evidence from DEMO processor business cases, as stated in section 8.9.8.

8.8. The first DEMO Processor Application in Production

In the year 2012, the first practical application of the DEMO processor has been implemented for a department of a public company that delivers energy and provides services. For that department, a tailor-made contract for each individual customer is the production to be delivered. The contracts are quite complex entities, and subject to business procedures derived from company policies, legal regulations and financial constraints. A high level of service to customers, like short response times is also required. The quality of the contracts should be high; errors are unacceptable. In addition incomplete transactions, deadlocks, deadline overflow must be eliminated. The contracts involve many documents from different sources, communication to subcontractors and communication to financial systems. This case is further discussed in Appendix IV.

8.9. Reflections on the $\Phi$, $\tau$ and $\Psi$ theories versus Bullshit

In his farewell speech in 2009 at the Delft University of Technology, Professor J.L.G. Dietz, developer of the theory of enterprise ontology and the GSDP, elaborated the foundations of his theories that are provided by the $\Phi$, $\tau$ and $\Psi$ theories. He stated that it is
either this, \( \Phi \) and \( \tau \) and \( \Psi \), and if it is not, then it is bullshit. Bullshit is a term used by the philosopher H.G. Frankfurt in his book “On Bullshit”, [1986]. It denotes the behavior of certain people who are not concerned about the truth or falseness of their statements but only use the statements for their own personal agenda. He states that those who are selling bullshit are even worse than those that are liars because liars are consistent. They lie also coherently and know what they are doing. Bullshitters are not consistent and don’t know what they are doing.

In the design of organizations there are many, hundreds, methodologies, chart drawing practices, frameworks being used by senior business consultants. Dietz states that these consultants are selling bullshit if their methodologies are not well founded on appropriate scientific theories. In fact, the current problem to design enterprises to implement a specific strategy is evident when empirical evidence shows that more than 90% of these initiatives fail. Two important types of shortcomings are usually immediately visible. The first being a lack of support for validation; there is no way to check if the proposed result comply with and explain the phenomena in reality, in our case enterprises. The second shortcoming is a lack of formal rigor of the proposed results. There may be ambiguity in interpretation, incompleteness, anomalies. These result are inadequate for any implementation in software.

The fact hat in many engineering disciplines we are so successful is due to the fact that we apply the right and appropriate scientific theories, mathematics and the engineering methodologies. There is validation and formal rigor. This is the case even when this is not so clearly visible anymore due to the use of advanced engineering tools.

Mathematics and formal methods are extremely powerful, mandatory to provide the needed formal rigor, but only suitable to support the internal consistence and verification purposes. Mathematics and formal methods are exclusively oriented at the objective white box perspective of a system. The subjective functional alignment of a system to functional requirements cannot be supported by objective mathematics and formal methods. In other words, mathematics and formal methods do not support any validation, which is so much needed to achieve functional appropriateness. Mathematics and formal methods are therefor mandatory and very good but explicitly not good enough for the development of any software systems that must deliver a good degree of functional appropriateness.

For functionally good software systems the Generic Systems Development Process (section 1.2.6) is required to deliver the functional and constructional perspectives on a system, to support the notion of architecture with the functional and the constructional principles. In successful mature engineering disciplines, the GSDP is almost hidden, not visible, but always implicitly applied. The using system ontology of the GSDP expresses our need to use an ontology with a high degree of appropriateness to fulfill our functional requirements. An ontology can only be appropriate if it is well founded on the empirical sciences.
Enterprise ontology, our using any domain ontology, must be founded on the empirical sciences. In our case, there is the intersubjective communication theory provided by Habermas [1981] and others, that is part of the $\Psi$ theory (section 3.6). We use the $\Psi$ theory to understand the world of phenomena, in this case individual human actors that cooperate and communicate etc. The $\Psi$ theory provides us with the US ontology of the GSDP and the GSDP-MDE methodology. Without the $\Psi$ theory, this thesis with the results for enterprise software engineering would not have been possible.

We use the $\Phi$ theory (section 1.2.3), which is one of the foundations of ontology, that provides us on one side a truthful and appropriate representation of the world of phenomena. On the other side, it provides us a formal representation using symbols or signs. Without the $\Phi$ theory, an ontology as the foundation of the developed GSDP-MDE methodology, this thesis with the results for enterprise software engineering would not have been possible.

We use the $\tau$ theory (section 1.2.4) which is also a foundation of the GSDP methodology that provides us with the relation between construction, engineering and implementation of systems. Without the $\tau$ theory and the GSDP methodology, the GSDP-MDE methodology of this thesis would not have been possible.

In science, there is one and only one scientific theory for any given domain. It is impossible to have two different theories for the same domain that are both appropriate. They must be identical or at least one of these theories is flawed. The appropriateness of a given scientific theory is delivered only by empirical evidence. There is now good empirical evidence that the $\Phi$, $\tau$ and $\Psi$ theories are appropriate. This implies that that other, different theories that are also appropriate, cannot exist.

It is now shown that for the design and construction of enterprises and enterprise information systems the $\Phi$, $\tau$ and $\Psi$ theories are the foundation. Without these theories, we would sell bullshit and we would know it.

8.10. Future Research Topics

In this section, future research topics are discussed regarding the professional application of the DEMO processor. Some topics address the relation and interfacing with other types of IT systems, so-called Management Information Systems$^{102}$ (MIS) and production support.

---

$^{102}$ MIS systems include for example financial accounting systems, value delivery systems and personnel systems.
8. Conclusions and Further Research

These topics are included for three reasons. The first is that DEMO models claim to offer a complete specification of the enterprise, including all produced instances of atomic fact and result types. The second is that the DEMO processor claims to implement DEMO models completely and should be capable to calculate all instances of fact and result types. The third is that MIS type of systems are faced with a very serious data-acquisition problem; how to acquire all relevant and truthful facts that are needed for an appropriate representation of the enterprise?

8.10.1. Temporal Logic Verification and Simulation

The correctness of the \(\Psi\) processor axioms has been investigated in Appendix I, using a pragmatic testing method including simulations. This provides a reasonable level of confidence of correctness, typically 'good enough' for 'normal' software applications. However, the requirements concerning correctness for the DEMO processor are much more stringent. Temporal logic verifications using simulations seems to be suitable for a proof of correctness of the DEMO processor.

8.10.2. Fine-grained DEMO Modeling Methodology

The DEMO models represent an ontological perspective, with exclusively ontological elements, according to the distinction axiom (section 2.5). The purpose is to minimize complexity as much as possible and separate the ontological level from implementation-specific details. However, for any implementation of an essential model, non-ontological transactions, the Infological (I-) and Datalogical (D-) transactions have to be designed and added to the ontological model of the B-organization. This is called fine-grained DEMO modeling.

8.10.3. Adaptive Model-instance Driven Case Management Systems

This research topic aims at the practical implementation of the fine-grained DEMO modeling methodology. The DEMO processor is a model-instance driven information system for sets of production instances. An interesting application domain is case management systems [OMG, 2009]. These are information systems for the production of production instances managed by the execution of instances of models. Case management systems include, in addition to the EIS, a software production environment for the pfacts of each transaction. The notion adaptive also refers to the capability to construct and destroy model components at runtime and replace these by newer and better ones. Assume that a transaction fails to

103 Production support systems include for example SCADA (supervisory control and data acquisition) systems, logistic systems etc.
deliver the requested production. Then this transaction instance must be destroyed. A new
transaction must be constructed with another model instance and optionally another actor-
executor. A reasonably satisfying informal methodology is available but needs fine-tuning
and proof of compliance with DEMO.

8.10.4.  Actor Role Mapping to Actors and Nested Actor Groups

DEMO models deliver precise specifications of the Actor world according to the Operation
Axiom (section 2.5). The Actor world specifies only actor roles. In practical production
environments, roles must be assigned to individual persons while so-called business
policies should be followed. The assignment of roles to individual human actors requires a
methodology and an implementation. The following situations may occur, governed by
business policies expressed in rules:

1.  Roles can be assigned to actors directly on a fixed basis.

2.  Roles can be assigned to actors where, if an actor does not respond within a limited
    period, the assignment will be made to a replacing actor, using the DEMO delegation
    mechanism.

3.  Roles can be assigned to groups of actors where one of the group members is
    assigned by some mechanism.

4.  Groups may be composed of individual actors and groups in a nested way.

The methodology to be developed should deliver:

1.  A set of primitives of business policy rules that is precisely expressive to model any
    imaginable set of business policy rules for role-actor mapping that may be used by
    enterprises.

2.  A grammar to construct aggregated business rules and algorithms to calculate role-
    actor assignments. These algorithms are to be implemented in a suitable software
    engine.

3.  A conceptual representation of actors, and nested actor groups for which these
    business policy rules can be applied.

4.  Further requirements are the support for rapid and flexible changes of the enterprise
    changes and simulation capabilities.

The business policy rules include for example: 1) a workload balancing mechanism as part
of the business policy calculation that would improve the practical value; and 2) an
assessment of competence, authority and responsibility of actors.

This mapping is closely related but not identical to the enterprise organization. Workload
measurements deliver valuable information for staff assessment.
8.10.5.  Interfacing to Management Information Systems

Large corporations use dedicated management information systems (MIS) for example for financial reporting that is directly derived from the ongoing production, direct and indirect costs, sales and purchase of raw materials over time. These MIS systems use so-called Value Delivery Models, based on high level concepts that are directly derived from accounting principles and complex legal regulations. An example is the REA ontology [Geerts, McCarthy, 2000] for financial systems. The strength of the REA approach is that a formal ontology is being used as foundation.

The correct operation of these systems is faced with two major problems:

i.) It is very difficult to obtain detailed, complete and consistent production and cost data of every department or subsidiary of the whole enterprise. This problem is caused by a lack of C4-ness information produced by the enterprise operating in reality.

ii.) The available information is typically expressed in some proprietary format and data primitives and delivered by the many different proprietary production systems. There is usually no correct conceptual mapping of the used primitives to the primitives used in the financial information systems.

This research topic involves the development of a methodology for the systematic mapping of enterprise ontology concepts to higher aggregated business concepts for MIS systems, in particular for financial information systems and value delivery systems.

8.10.6.  Alignment of DEMO and Normalized Systems

The approach in this research to address the problem statement of section 1 is the use of high level conceptual models, expressed in an ontological language. The third problem mentioned in section 1 is the increasing costs of software maintenance, due to increasing complexity over time.

The benefits of DEMO processor based systems to reduce complexity are:

i.)  For large and complex enterprises, there is a large and complex information processing problem domain. For this domain, information systems have to be designed and implemented.

ii.) From the original large and complex domain of information-processing all transactions, actors and communication are isolated and abstracted. For this domain, there is a high quality approach and solution available.

iii.) The remaining domain of information processing to be solved is now smaller in volume, less complex and better structured. However, this domain demands also a high quality approach for information systems engineering, which is not supplied by DEMO.
8. Conclusions and Further Research

iv.) From a systems theory point of view DEMO addresses the white box engineering of an enterprise. DEMO does not deliver any black box specifications and is software implementation independent.

The theory of Normalized Systems addresses also the third fundamental problem in software engineering mentioned in section 1, the combinatorial explosion of complexity of software systems during subsequent modifications. After each modification, the software system becomes more unmanageable in terms of complexity and costs. The high failure rate of IT projects is partly due to the phenomenon that the internal structure of the IT systems has become unstable, "worn out" or deteriorated even before going into production. Current generations of software systems, based on current architectures, fundamentally resist change. This phenomenon has a deeper root, the increasing entropy of a system over time, expressed by the second law of thermodynamics. Normalized Systems (NS) addresses stability problems by focusing on the modularity of software 'pieces' or components. There are design principles, expressed in four theorems. Normalized Systems is explicitly independent from languages, frameworks, and notations and is applicable to virtually all application development environments and applications.

An informal assessment of DEMO against the four theorems of Normalized systems delivers the conjecture that DEMO models and the DEMO processor are Normalized systems compliant. The zero entropy of DMOL models expressed in DMOL support this conjecture. These considerations and reflections require further research.

A pragmatic stance is that the GSDP-MDE approach provides an executable system for any EIS. Any non-EIS systems should be implemented in NS compliant application development environments to address the software complexity problem.

8.10.7. Enhancing BPMN Modeling Methodology using Enterprise Ontology

As shown before, the DEMO processor calculates automatically with formal correctness (to be confirmed by the results of the temporal logic simulations, section 8.9.1) and ontological completeness the communication patterns between enterprise actors and the coordination from DEMO models. In [Van Nuffel, 2009] the limitations of BPMN have been addressed. These limitations are solved by the DEMO processor. It is a conjecture that DEMO modeling should be the foundation of business process modeling. This approach may result in a strong formal foundation for BPMN. The advantages of BPMN, the wide acceptance and knowledge base in the professional world may support acceptance.

The main advantages but also potential enhancements of DEMO based enterprise control compared to BPMN workflow are:
8. Conclusions and Further Research

i.) Concerning ontological completeness, BPMN lacks transaction completeness, where DEMO models under execution exhibit perfectly calculated transaction completeness and lack of anomalies such as deadlocks.

ii.) DEMO models offer the complete production and communication fact types. Calculations of all fact type instances can be exported to external information systems.

iii.) DEMO models and the DMOL language exhibit no construct anomalies like:

- Construct excess\textsuperscript{104},
- construct overload\textsuperscript{105}, and,
- Construct redundancy\textsuperscript{106}.

\textsuperscript{104} Construct excess refers to a construct that does not relate to ontological artifacts.
\textsuperscript{105} Construct overload refers to a construct that relates to more than one ontological artifact.
\textsuperscript{106} Construct redundancy refers to more than one construct that relate to one ontological artifact.
8.10.8. Quality assessment of Business-IT alignment of DEMO processor EISs

It has been argued that one of the main problems of information systems for enterprises is the lack of business-IT alignment; the question to which degree an information systems supports the operation of an enterprise. Intensive modeling sessions with customers about their enterprise is the current way to optimize business-IT alignment. However, the actual improvements have to be demonstrated also.

A quality assessment has to be made for state of the art IT systems versus DEMO-processor based IT systems based on a statistical significant number of case studies. For the quality assessment, dedicated methodologies and metrics have to be developed to measure, quantify and compare outcomes. As there are currently no significant number of DEMO processor IT systems in full production, no empirical claims can be made.
PART VI

References and Appendixes
References


References


Ciao! Consortium; Cooperation & Interoperability - Architecture & Ontology, online: www.ciaonetwork.org


Frankfurt, H.G.: On Bullshit. Raritan Quarterly Review 6, no. 2 (Fall 1986)


References


Halpin, T.: Object Role Modeling (ORM). online: http://www.orm.net/


ITGI, The IT Governance Institute. online: http://www.itgi.org/


OMG, Model Driven Architecture Specifications, The OMG. (2001)


References


SIKS Dissertations

1998
1998-1 Johan van den Akker (CWI), DEGAS - An Active, Temporal Database of Autonomous Objects
1998-2 Floris Wiesman (UM), Information Retrieval by Graphically Browsing Meta-Information
1998-3 Ans Steuten (TUD), A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective
1998-4 Dennis Breuker (UM), Memory versus Search in Games
1998-5 E.W.Oskamp (RUL), Computerondersteuning bij Straftoemeting

1999
1999-1 Mark Sloof (VU), Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products
1999-2 Rob Potharst (EUR), Classification using decision trees and neural nets
1999-3 Don Beal (UM), The Nature of Minimax Search
1999-4 Jacques Penders (UM), The practical Art of Moving Physical Objects
1999-5 Aldo de Moor (KUB), Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems
1999-6 Niek J.E. Wijngaards (VU), Re-design of compositional systems
1999-7 David Spelt (UT), Verification support for object database design
1999-8 Jacques H.J. Lenting (UM), Informed Gambling: Conception and Analysis of a Multi-Agent, Mechanism for Discrete Reallocation.

2000
2000-1 Frank Niessink (VU), Perspectives on Improving Software Maintenance.
2000-2 Koen Holtman (TUE), Prototyping of CMS Storage Management
2000-3 Carollen M.T. Metselaar (UVA), Sociaal-organisatorische gevolgen van kennisniettechnologie; een procesbenadering en actorperspectief.
2000-4 Geert de Haan (VU), ETAG, A Formal Model of Competence Knowledge for User Interface Design
2000-5 Ruud van der Pol (UM), Knowledge-based Query Formulation in Information Retrieval.
2000-6 Rogier van Eijk (UU), Programming Languages for Agent Communication
2000-7 Niels Peek (UU), Decision-theoretic Planning of Clinical Patient Management
2000-8 Veerle Coup (EUR), Sensitivity Analysis of Decision-Theoretic Networks
2000-9 Florian Waas (CWI), Principles of Probabilistic Query Optimization
2000-10 Niels Nes (CWI), Image Database Management System Design Considerations, Algorithms and Architecture
2000-11 Jonas Karlsson (CWI), Scalable Distributed Data Structures for Database Management

2001
2001-1 Silja Renooij (UU), Qualitative Approaches to Quantifying Probabilistic Networks
2001-2 Koen Hindriks (UU), Agent Programming Languages: Programming with Mental Models,
2001-3 Maarten van Someren (UvA), Learning as problem solving
2001-4 Evgueni Smirnov (UM), Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets
2001-5 Jacco van Ossenbruggen (VU), Processing Structured Hypermedia: A Matter of Style
2001-6 Martin van Welie (VU), Task-based User Interface Design
2001-7 Bastiaan Schoonghage (VU), Diva: Architectural Perspectives on Information Visualization
2001-8 Pascal van Eck (VU), A Compositional Semantic Structure for Multi-Agent Systems Dynamics.
2001-9 Pieter Jan ‘t Hoen (RUL), Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes
2001-10 Maarten Sierhuis (UvA), Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design.
2001-11 Tom M. van Engers (VUA), Knowledge Management: The Role of Mental Models in Business Systems Design

2002
2002-01 Nico Lassing (VU), Architecture-Level Modifiability Analysis
References

2002-02 Roelof van Zwol (UT), Modelling and searching web-based document collections
2002-03 Henk Ernst Blok (UT), Database Optimization Aspects for Information Retrieval
2002-04 Juan Roberto Castelo Valdueza (UU), The Discrete Acyclic Digraph Markov Model in Data Mining
2002-05 Radu Serban (VU), The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents
2002-06 Laurens Mommers (UL), Applied legal epistemology; Building a knowledge-based ontology of the legal domain
2002-07 Peter Boncz (CWI), Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications
2002-08 Jaap Gordijn (VU) Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas
2002-09 Willems-Jan van den Heuvel(KUB) Integrating Modern Business Applications with Objectified Legacy Systems
2002-10 Brian Sheppard (UM), Towards Perfect Play of Scrabble
2002-11 Wouter C.A. Wijngaards (VU), Agent Based Modelling of Dynamics: Biological and Organisational Applications
2002-12 Albrecht Schmidt (Uva) Processing XML in Database Systems
2002-13 Hongjing Wu (TUE), A Reference Architecture for Adaptive Hypermedia Applications
2002-14 Wieke de Vries (UU), Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems
2002-15 Rik Eshuis (UT), Semantics and Verification of UML Activity Diagrams for Workflow Modelling
2002-17 Stefan Manegold (UVA), Understanding, Modeling, and Improving Man-Memory Database Performance

2003
2003-01 Heiner Stuckenschmidt (VU), Ontology-Based Information Sharing in Weakly Structured Environments
2003-02 Jan Broersen (VU), Modal Action Logics for Reasoning About Reactive Systems
2003-03 Marjon Schuermie (TUD), Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy
2003-04 Milan Petkovic (UT), Content-Based Video Retrieval Supported by Database Technology
2003-05 Jos Lehmann (UVA) Causation in Artificial Intelligence and Law - A modelling approach
2003-06 Boris van Schooten (UT), Development and specification of virtual environments
2003-07 Machiel Jansen (UvA), Formal Explorations of Knowledge Intensive Tasks
2003-08 Yongping Ran (UM), Repair Based Scheduling
2003-09 Rens Kortmann (UM), The resolution of visually guided behaviour
2003-10 Andreas Lincke (UvT), Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture
2003-11 Simon Keizer (UT), Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks
2003-12 Roeland Ordelman (UT), Dutch speech recognition in multimedia information retrieval
2003-13 Jeroen Donkers (UM), Nosce Hostem - Searching with Opponent Models
2003-14 Stijn Hoppenbrouwers (KUN), Freezing Language: Conceptualisation Processes across ICT-Supported Organisations
2003-15 Mathijs de Weerdt (UvT) Plan Merging in Multi-Agent Systems
2003-16 Menzo Windhouwer (CWI) Feature Grammar Systems - Incremental Maintenance of Indexes to igital Media Warehouses
2003-17 David Jansen (UT), Extensions of Statecharts with Probability, Time, and Stochastic Timing
2003-18 Levente Kocsis (UM), Learning Search Decisions

2004
2004-01 Virginia Dignum (UU), A Model for Organizational Interaction: Based on Agents, Founded in Logic
2004-02 Lai Xu (UvT), Monitoring Multi-party Contracts for E-business
2004-03 Perry Groot (VU) A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving
2004-04 Chris van Aart (UVA) Organizational Principles for Multi-Agent Architectures
2004-05 Viara Popova (EUR) Knowledge discovery and monotonicity
2004-07 Elise Boltes (UM) Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes
2004-08 Joop Verbeek(UM) Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politie en de gegevensuitwisseling en digitale expertise
2004-09 Martin Caminada (UU) For the Sake of the Argument; explorations into argument-based reasoning
2004-10 Suzanne Kabel (UVA) Knowledge-rich indexing of learning-objects
2004-11 Michel Klein (VU) Change Management for Distributed Ontologies
2004-12 The Duy Bui (UT) Creating emotions and facial expressions for embodied agents
2004-13 Wojciech Jamroga (UT) Using Multiple Models of Reality: On Agents who Know how to Play
2004-14 Paul Hamerstein (UU) Logic in Conflict. Logical Explorations in Strategic Equilibrium
2004-15 Arno Knobbe (UU) Multi-Relational Data Mining
2004-16 Federico Divina (VU) Hybrid Genetic Relational Search for Inductive Learning
2004-17 Mark Winands (UM) Informed Search in Complex Games
2004-18 Vania Bessa Machado (UvA) Supporting the Construction of Qualitative Knowledge Models
2004-19 Thijis Westerveld (UT) Using generative probabilistic models for multimedia retrieval
2004-20 Madelon Evers (Nyenrode) Learning from Design: facilitating multidisciplinary design teams

2005
2005-01 Floor Verdenius (UVA) Methodological Aspects of Designing Induction-Based Applications
2005-02 Erik van der Werf (UM) AI techniques for the game of Go
2005-03 Franc Grootjens (RUN) A Pragmatic Approach to the Conceptualisation of Language
2005-04 Nirvana Meratnia (UT) Towards Database Support for Moving Object data
2005-05 Gabriel Infante-Lopez (UVA) Two-Level Probabilistic Grammars for Natural Language Parsing
2005-06 Pieter Spronck (UM) Adaptive Game AI
2005-07 Flavius Frasincar (TUE) Hypermedia Presentation Generation for Semantic Web Information Systems
2005-08 Richard Vdovjak (TUE) A Model-driven Approach for Building Distributed Ontology-based Web Applications
2005-09 Jeroen Broekstra (VU) Storage, Querying and Inferencing for Semantic Web Languages
2005-10 Thijis Westerveld (UT) Using generative probabilistic models for multimedia retrieval
2005-11 Thijis Westerveld (UT) Using generative probabilistic models for multimedia retrieval
2005-12 Csaba Boer (EUR) Distributed Simulation in Industry
2005-13 Fred Hamburg (UL) Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen
2005-14 Boris Omelayenko (VU) Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics
2005-15 Tibor Bosse (VU) Analysis of the Dynamics of Cognitive Processes
2005-16 Joris Graaumans (UU) Usability of XML Query Languages
2005-17 Boris Shishkov (TUD) Software Specification Based on Re-usable Business Components
2005-18 Danielle Sent (UU) Test-selection strategies for probabilistic networks
2005-19 Michel van Dartel (UM) Situated Representation
2005-20 Cristina Coteanu (UL) Cyber Consumer Law, State of the Art and Perspectives
2005-21 Wijnand Derks (UT) Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006
2006-01 Samuil Angelov (TUE) Foundations of B2B Electronic Contracting
2006-02 Cristina Chisalita (VU) Contextual issues in the design and use of information technology in organizations
2006-03 Noor Christoph (UVA) The role of metacognitive skills in learning to solve problems
2006-04 Manda Sabou (VU) Building Web Service Ontologies
2006-05 Eelco Herder (UT) Forward, Back and Home Again - Analyzing User Behavior on the Web
2006-06 Mohamed Wahdan (UM) Automatic Formulation of the Auditor's Opinion
2006-07 Rainer Malik (UU) CONAN: Text Mining in the Biomedical Domain
2006-08 Ronny Siebes (VU) Semantic Routing in Peer-to-Peer Systems
2006-09 Thijis Westerveld (UT) Flatting Queries over Nested Data Types
2006-10 Reiner Malik (UU) CONAN: Text Mining in the Biomedical Domain
2006-11 Johan Hoorn (VU) Software Requirements: Update, Upgrade, Redesign – towards a Theory of Requirements Change
2006-12 Bert Bongers (VU) Interaction - Towards an e-cology of people, our technological environment, and the arts
2006-13 Jeroen van Ruth (UT) Flattening Queries over Nested Data Types
2006-14 Stacey Nagata (UU) User Assistance for Multitasking with Interruptions on a Mobile Device
2006-15 Henk-Jan Lebbink (UU) Dialogue and Decision Games for Information Exchanging Agents
2006-16 Carsten Riggelsen (UU) Approximation Methods for Efficient Learning of Bayesian Networks
2006-17 Thijs Westerveld (UT) Using generative probabilistic models for multimedia retrieval
2006-18 Valentin Zhizhikun (UVA) Graph transformation for Natural Language Processing
References

2006-20 Marina Velikova (UvT) Monotone models for prediction in data mining
2006-21 Bas van Gils (RUN) Aptness on the Web
2006-22 Paul de Vrieze (RUN) Fundamentals of Adaptive Personalisation
2006-23 Ion Juvina (UU) Development of Cognitive Model for Navigating on the Web
2006-24 Laura Hollink (VU) Semantic Annotation for Retrieval of Visual Resources
2006-25 Madalina Drugan (UU) Conditional log-likelihood MDL and Evolutionary MCMC
2006-26 Vojkan Mihajlovic (UT) Score Region Algebra: A Flexible Framework for Structured Information Retrieval
2006-27 Stefano Bocconi (CWI) Vox Populi: generating video documentaries from semantically annotated media repositories
2006-28 Borkur Sigurbjornsson (UVA) Focused Information Access using XML Element Retrieval

2007
2007-01 Kees Leune (UvT) Access Control and Service-Oriented Architectures
2007-02 Wouter Teepe (RUG) Reconciling Information Exchange and Confidentiality: A Formal Approach
2007-03 Peter Mika (VU) Social Networks and the Semantic Web
2007-06 Glad Mishne (UVA) Applied Text Analytics for Blogs
2007-07 Natasa Jovanovic’ (UT) To Whom It May Concern - Addressee Identification in Face-to-Face Meetings
2007-08 Mark Hoogendoorn (VU) Modeling of Change in Multi-Agent Organizations
2007-09 David Mobach (VU) Agent-Based Mediated Service Negotiation
2007-11 Natalia Stash (TUE) Social Networks and the Semantic Web
2007-12 Marcel van Gerven (RUN) Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty
2007-13 Rutger Rienks (UT) Meetings in Smart Environments; Implications of Progressing Technology
2007-14 Niek Bergboer (UM) Context-Based Image Analysis
2007-15 Joost Schalken (VU) NIM: a Situated Computational Memory Model
2007-16 Davide Grossi (UU) Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems
2007-18 Bart Orriens (UvT) On the development of an infrastructure of adaptive business collaborations
2007-19 David Levy (UM) Intimate relationships with artificial partners
2007-21 Karianne Vermaas (UU) Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005
2007-22 Zlatko Zlatev (UT) Goal-oriented design of value and process models from patterns
2007-23 Peter Barna (TUE) Specification of Application Logic in Web Information Systems
2007-24 Georgina Ramirez Camps (CWI) Structural Features in XML Retrieval

2008
2008-01 Katalin Boer-Sorban (EUR) Agent-Based Simulation of Financial Markets: A modular, continuous-time approach
2008-03 Vera Hollink (UVA) Optimizing hierarchical menus: a usage-based approach
2008-04 Ander de Keijzer (UT) Management of Uncertain Data - towards unattended integration
2008-05 Bela Mutschler (UT) Modeling and simulating causal dependencies on process-aware information systems from a cost perspective
2008-06 Arjen Hommersom (RUN) On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective
2008-07 Peter van Rosmalen (OU) Supporting the tutor in the design and support of adaptive e-learning
2008-08 Janneke Bolt (UU) Bayesian Networks: Aspects of Approximate Inference
2008-09 Christof van Nimwegen (UU) The paradox of the guided user: assistance can be counter-effective
<table>
<thead>
<tr>
<th>Year</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-10</td>
<td>Wauter Bosma (UT)</td>
<td>Discourse oriented summarization</td>
</tr>
<tr>
<td>2008-11</td>
<td>Vera Kartseva (VU)</td>
<td>Designing Controls for Network Organizations: A Value-Based Approach</td>
</tr>
<tr>
<td>2008-12</td>
<td>Jozsef Farkas (RUN)</td>
<td>A Semiotically Oriented Cognitive Model of Knowledge Representation</td>
</tr>
<tr>
<td>2008-13</td>
<td>Caterina Carraciolo (UVA)</td>
<td>Topic Driven Access to Scientific Handbooks</td>
</tr>
<tr>
<td>2008-14</td>
<td>Arthur van Bunningen (UT)</td>
<td>Context-Aware Querying; Better Answers with Less Effort</td>
</tr>
<tr>
<td>2008-16</td>
<td>Henriette van Vugt (VU)</td>
<td>Embodied agents from a user's perspective</td>
</tr>
<tr>
<td>2008-17</td>
<td>Martin Op ´t Land (TUD)</td>
<td>Applying Architecture and Ontology to the Splitting and Allying of Enterprises</td>
</tr>
<tr>
<td>2008-18</td>
<td>Guido de Croon (UM)</td>
<td>Adaptive Active Vision</td>
</tr>
<tr>
<td>2008-19</td>
<td>Henning Rode (UT)</td>
<td>From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search</td>
</tr>
<tr>
<td>2008-20</td>
<td>Rex Arendsen (UVA)</td>
<td>Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven</td>
</tr>
<tr>
<td>2008-21</td>
<td>Krisztián Balog (UVA)</td>
<td>People Search in the Enterprise</td>
</tr>
<tr>
<td>2008-22</td>
<td>Henk Koning (UU)</td>
<td>Communication of IT-Architecture</td>
</tr>
<tr>
<td>2008-23</td>
<td>Zharko Aleksovski (VU)</td>
<td>Using background knowledge in ontology matching</td>
</tr>
<tr>
<td>2008-26</td>
<td>Ildiko Flesch (RUN)</td>
<td>On the Use of Independence Relations in Bayesian Networks</td>
</tr>
<tr>
<td>2008-27</td>
<td>Dennis Reidsma (UT)</td>
<td>From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search</td>
</tr>
<tr>
<td>2008-28</td>
<td>Frank Terpstra (UVA)</td>
<td>Scientific Workflow Design; theoretical and practical issues</td>
</tr>
<tr>
<td>2008-29</td>
<td>Jeroen de Knijf (UU)</td>
<td>Studies in Frequent Tree Mining</td>
</tr>
<tr>
<td>2008-30</td>
<td>Sietse Overbeek (RUN)</td>
<td>Bridging Supply and Demand for Knowledge Intensive Tasks – Based on Knowledge, Cognition, and Quality</td>
</tr>
<tr>
<td>2008-31</td>
<td>Muhammad Subianto (UU)</td>
<td>Understanding Classification</td>
</tr>
<tr>
<td>2008-32</td>
<td>Ronald Poppe (UT)</td>
<td>Discriminative Vision-Based Recovery and Recognition of Human Motion</td>
</tr>
<tr>
<td>2008-33</td>
<td>Hans Stol (UU)</td>
<td>A Framework for Evidence-based Policy Making Using IT</td>
</tr>
<tr>
<td>2008-34</td>
<td>Josephine Nabukwasi (RUN)</td>
<td>Improving the Quality of Organisational Policy Making using Collaboration Engineering</td>
</tr>
<tr>
<td>2008-35</td>
<td>Sietse Overbeek (RUN)</td>
<td>Bridging Supply and Demand for Knowledge Intensive Tasks – Based on Knowledge, Cognition, and Quality</td>
</tr>
<tr>
<td>2009</td>
<td>Rasa Jurgelenaite (RUN)</td>
<td>Symmetric Causal Independence Models</td>
</tr>
<tr>
<td>2009-01</td>
<td>Willem Robert van Hage (VU)</td>
<td>Evaluating Ontology-Alignment Techniques</td>
</tr>
<tr>
<td>2009-02</td>
<td>Hans Stol (UU)</td>
<td>A Framework for Evidence-based Policy Making Using IT</td>
</tr>
<tr>
<td>2009-03</td>
<td>Josephine Nabukwasi (RUN)</td>
<td>Improving the Quality of Organisational Policy Making using Collaboration Engineering</td>
</tr>
<tr>
<td>2009-04</td>
<td>Alexander Boer (UVA)</td>
<td>Legal Theory, Sources of Law &amp; the Semantic Web</td>
</tr>
<tr>
<td>2009-12</td>
<td>Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin)</td>
<td>operating Guidelines for Services</td>
</tr>
<tr>
<td>2009-13</td>
<td>Steven de Jong (UM)</td>
<td>Fairness in Multi-Agent Systems</td>
</tr>
<tr>
<td>2009-14</td>
<td>Maksym Korotkiy (VU)</td>
<td>From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)</td>
</tr>
<tr>
<td>2009-15</td>
<td>Rinke Hoekstra (UVA)</td>
<td>Ontology Representation - Design Patterns and Ontologies that Make Sense</td>
</tr>
<tr>
<td>2009-16</td>
<td>Fritz Reul (UvT)</td>
<td>New Architectures in Computer Chess</td>
</tr>
<tr>
<td>2009-17</td>
<td>Laurens van der Maaten (UvT)</td>
<td>Feature Extraction from Visual Data</td>
</tr>
<tr>
<td>Year</td>
<td>Reference</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>2009-18</td>
<td>Fabian Groffen (CWI) Armada, An Evolving Database System</td>
<td></td>
</tr>
<tr>
<td>2009-19</td>
<td>Valentin Robu (CWI) Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets</td>
<td></td>
</tr>
<tr>
<td>2009-20</td>
<td>Bob van der Vecht (UU) Adjustable Autonomy: Controlling Influences on Decision Making</td>
<td></td>
</tr>
<tr>
<td>2009-21</td>
<td>Stijn Vanderlooy (UM) Ranking and Reliable Classification</td>
<td></td>
</tr>
<tr>
<td>2009-22</td>
<td>Pavel Serdyukov (UT) Search For Expertise: Going beyond direct evidence</td>
<td></td>
</tr>
<tr>
<td>2009-23</td>
<td>Peter Hofgesang (VU) Modelling Web Usage in a Changing Environment</td>
<td></td>
</tr>
<tr>
<td>2009-24</td>
<td>Annerieke Heuvelink (VUA) Cognitive Models for Training Simulations</td>
<td></td>
</tr>
<tr>
<td>2009-25</td>
<td>Alex van Ballegooij (CWI) &quot;RAM: Array Database Management through Relational Mapping&quot;</td>
<td></td>
</tr>
<tr>
<td>2009-26</td>
<td>Fernando Koch (UU) An Agent-Based Model for the Development of Intelligent Mobile Services</td>
<td></td>
</tr>
<tr>
<td>2009-27</td>
<td>Christian Glahn (OU) Contextual Support of collaborative Engagement and Reflection on the Web</td>
<td></td>
</tr>
<tr>
<td>2009-28</td>
<td>Sander Evers (UT) Sensor Data Management with Probabilistic Models</td>
<td></td>
</tr>
<tr>
<td>2009-29</td>
<td>Stanislav Pokraev (UT) Model-Driven Semantic Integration of Service-Oriented Applications</td>
<td></td>
</tr>
<tr>
<td>2009-30</td>
<td>Marcin Zukowski (CWI) Balancing vectorized query execution with bandwidth-optimized storage</td>
<td></td>
</tr>
<tr>
<td>2009-31</td>
<td>Sofiya Katrenko (UVA) A Closer Look at Learning Relations from Text</td>
<td></td>
</tr>
<tr>
<td>2009-32</td>
<td>Rik Farenhorst (VU) and Remco de Boer (VU) Architectural Knowledge Management: Supporting Architects and Auditors</td>
<td></td>
</tr>
<tr>
<td>2009-33</td>
<td>Khiet Truong (UT) How Does Real Affect Affect Recognition In Speech?</td>
<td></td>
</tr>
<tr>
<td>2009-34</td>
<td>Inge van de Weerd (UU) Advancing in Software Product Management: An Incremental Method Engineering Approach</td>
<td></td>
</tr>
<tr>
<td>2009-35</td>
<td>Wouter Koelewijn (UL) Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling</td>
<td></td>
</tr>
<tr>
<td>2009-36</td>
<td>Marco Kalz (OUN) Placement Support for Learners in Learning Networks</td>
<td></td>
</tr>
<tr>
<td>2009-37</td>
<td>Hendrik Drachsler (OUN) Navigation Support for Learners in Informal Learning Networks</td>
<td></td>
</tr>
<tr>
<td>2009-38</td>
<td>Riina Vuorikari (OU) Tags and self-organisation: a metadata ecology for learning resources in a multilingual context</td>
<td></td>
</tr>
<tr>
<td>2009-40</td>
<td>Stephan Raaijmakers (UvT) Multinomial Language Learning: Investigations into the Geometry of Language</td>
<td></td>
</tr>
<tr>
<td>2009-41</td>
<td>Igor Berezhnyy (UvT) Digital Analysis of Paintings</td>
<td></td>
</tr>
<tr>
<td>2009-42</td>
<td>Toine Bogers Recommender Systems for Social Bookmarking</td>
<td></td>
</tr>
<tr>
<td>2009-43</td>
<td>Virginia Nunes Leal Franqueira (UT) Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Environments</td>
<td></td>
</tr>
<tr>
<td>2009-44</td>
<td>Roberto Santana Tapia (UT) Assessing Business-IT Alignment in Networked Organizations</td>
<td></td>
</tr>
<tr>
<td>2009-45</td>
<td>Jilles Vreeken (UU) Making Pattern Mining Useful</td>
<td></td>
</tr>
<tr>
<td>2009-46</td>
<td>Loredana Afanasiev (UvA) Querying XML: Benchmarks and Recursion</td>
<td></td>
</tr>
</tbody>
</table>

2010

2010-01 | Matthijs van Leeuwen (UU) Patterns that Matter |
| 2010-02 | Ingo Wassink (UT) Work flows in Life Science |
| 2010-04 | Olga Kulyk (OU) Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments |
| 2010-05 | Claudia Hauff (UT) Predicting the Effectiveness of Queries and Retrieval Systems |
| 2010-06 | Sander Bakkes (UvT) Rapid Adaptation of Video Game AI |
| 2010-07 | Wim Fikker (UT) Gesture interaction at a Distance |
| 2010-08 | Krzysztof Siewicz (UL) Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments |
| 2010-09 | Hugo Kielman (UL) A Politieke gegevensverwerking en Privacy, Naar een effectieve waarborging |
| 2010-10 | Rebecca Ong (UL) Mobile Communication and Protection of Children |
| 2010-11 | Adriaan Ter Borg (TUD) The world according to MARP: Multi-Agent Route Planning |
| 2010-12 | Susan van den Braak (UU) Sensemaking software for crime analysis |
| 2010-13 | Gianluigi Folino (RUN) High Performance Data Mining using Bio-inspired techniques |
| 2010-14 | Sander van Splunter (VU) Automated Web Service Reconfiguration |
| 2010-15 | Lianne Bodenstaff (UT) Managing Dependency Relations in Inter-Organizational Models |
| 2010-16 | Sicco Verwer (TUD) Efficient Identification of Timed Automata, theory and practice |
| 2010-17 | Spyros Kotoulas (VU) Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications |
2010-18 Charlotte Gerritsen (VU) Caught in the Act: Investigating Crime by Agent-Based Simulation
2010-19 Henriette Cramer (UvA) People’s Responses to Autonomous and Adaptive Systems
2010-20 Ivo Swartjes (UT) Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative
2010-21 Harold van Heerde (UT) Privacy-aware data management by means of data degradation
2010-22 Michiel Hildebrand (CWI) End-user Support for Access to Heterogeneous Linked Data
2010-23 Bas Steunebrink (UU) The Logical Structure of Emotions
2010-24 Dmytro Tykhonov Designing Generic and Efficient Negotiation Strategies
2010-25 Zulfiqar Ali Memon (VU) Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective
2010-26 Ying Zhang (CWI) XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines
2010-27 Marten Voulou (UL) Automatisch contracteren
2010-28 Arne Koopman (UU) Characteristic Relational Patterns
2010-29 Stratos Idreos (CWI) Database Cracking: Towards Auto-tuning Database Kernels
2010-30 Marieke van Erp (UvT) Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval
2010-31 Victor de Boer (UvA) An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems
2010-32 Marcell Hiel (UvT) An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems
2010-33 Robin Aly (UT) Modeling and Reasoning about Trust, Reputation, and Truth in Multi-agent Systems
2010-34 Teduh Dirghayu (UT) Interaction Design in Service Compositions
2010-35 Jose Janssen (OU) Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification
2010-36 Niels Lohmann (TUE) Correctness of services and their composition
2010-37 Dirk Fahland (TUE) From Scenarios to components
2010-38 Ghazanfar Farooq Siddiqui (VU) Integrative modeling of emotions in virtual agents
2010-39 Jan Martijn van der Werf (TUE) Compositional Design and Verification of Component-Based Information Systems
2010-40 Hado van Hasselt (UU) Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference learning algorithms
2010-41 Guillaume Chaslot (UM) Monte-Carlo Tree Search
2010-42 Yujia Cao (UT) Multimodal Information Presentation for High Load Human Computer Interaction
2010-43 Pieter-Paul van Maanen (VU) Adaptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention
2010-44 Nieske Vergunst (UU) BDI-based Generation of Robust Task-Oriented Dialogues
2010-45 Vasilios Andrikopoulos (UvT) A theory and model for the evolution of software services
2010-46 Vincent Pijpers (VU) e3alignment: Exploring Inter-Organizational Business-ICT Alignment
2010-47 Yanbing Fu (UT) Mining Process Model Variants: Challenges, Techniques, Examples
2010-48 Milan Lovric (EU) Behavioral Finance and Agent-Based Artificial Markets
2010-49 Jahn-Takeshi Salto (UM) Solving difficult game positions
2010-50 Bouke Huurnink (UVA) Search in Audiovisual Broadcast Archives
2010-51 Alia Khairia Amin (CU) Understanding and supporting information seeking tasks in multiple sources
2010-52 Peter van Maanen (VU) Adaptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention
2011-01 Botond Cseke (RUN) Variational Algorithms for Bayesian Inference in Latent Gaussian Models
2011-02 Nick Tinne Riemer (U) Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language
2011-03 Jan Martijn van der Werf (TUE) Compositional Design and Verification of Component-Based Information Systems
2011-04 Hado van Hasselt (UU) Insights in Reinforcement Learning: Formal analysis and empirical evaluation of temporal-difference learning algorithms
2011-05 Bas van der Raadt (VU) Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline
2011-06 Ywen Wang (TUE) Semantically-Enhanced Recommendations in Cultural Heritage
2011-07 Yuju Cao (UT) Multimodal Information Presentation for High Load Human Computer Interaction
2011-08 Nieske Vergunst (U) BDI-based Generation of Robust Task-Oriented Dialogues

References

2010-18 Charlotte Gerritsen (VU) Caught in the Act: Investigating Crime by Agent-Based Simulation
2010-19 Henriette Cramer (UvA) People’s Responses to Autonomous and Adaptive Systems
2010-20 Ivo Swartjes (UT) Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative
2010-21 Harold van Heerde (UT) Privacy-aware data management by means of data degradation
2010-22 Michiel Hildebrand (CWI) End-user Support for Access to Heterogeneous Linked Data
2010-23 Bas Steunebrink (UU) The Logical Structure of Emotions
2010-24 Dmytro Tykhonov Designing Generic and Efficient Negotiation Strategies
2010-25 Zulfiqar Ali Memon (VU) Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective
2010-26 Ying Zhang (CWI) XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines
2010-27 Marten Voulou (UL) Automatisch contracteren
2010-28 Arne Koopman (UU) Characteristic Relational Patterns
2010-29 Stratos Idreos (CWI) Database Cracking: Towards Auto-tuning Database Kernels
2010-30 Marieke van Erp (UvT) Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval
2010-31 Victor de Boer (UvA) An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems
2010-32 Marcell Hiel (UvT) An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems
2010-33 Robin Aly (UT) Modeling and Reasoning about Trust, Reputation, and Truth in Multi-agent Systems
2010-34 Teduh Dirghayu (UT) Interaction Design in Service Compositions
2010-35 Jose Janssen (OU) Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification
2010-36 Niels Lohmann (TUE) Correctness of services and their composition
2010-37 Dirk Fahland (TUE) From Scenarios to components
2010-38 Ghazanfar Farooq Siddiqui (VU) Integrative modeling of emotions in virtual agents
2010-39 Jan Martijn van der Werf (TUE) Compositional Design and Verification of Component-Based Information Systems
2010-40 Hado van Hasselt (UU) Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference learning algorithms
2010-41 Guillaume Chaslot (UM) Monte-Carlo Tree Search
2010-42 Yujia Cao (UT) Multimodal Information Presentation for High Load Human Computer Interaction
2010-43 Pieter-Paul van Maanen (VU) Adaptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention
2010-44 Nieske Vergunst (UU) BDI-based Generation of Robust Task-Oriented Dialogues

2011-01 Botond Cseke (RUN) Variational Algorithms for Bayesian Inference in Latent Gaussian Models
2011-02 Nick Tinne Riemer (U) Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language
2011-03 Jan Martijn van der Werf (TUE) Compositional Design and Verification of Component-Based Information Systems
2011-04 Hado van Hasselt (UU) Insights in Reinforcement Learning: Formal analysis and empirical evaluation of temporal-difference learning algorithms
2011-05 Bas van der Raadt (VU) Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline
2011-06 Ywen Wang (TUE) Semantically-Enhanced Recommendations in Cultural Heritage
2011-07 Yuju Cao (UT) Multimodal Information Presentation for High Load Human Computer Interaction
2011-08 Nieske Vergunst (U) BDI-based Generation of Robust Task-Oriented Dialogues
<table>
<thead>
<tr>
<th>Year</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>Tim de Jong (OU)</td>
<td>Contextualised Mobile Media for Learning</td>
</tr>
<tr>
<td>2011</td>
<td>Bart Bogaert (UV)</td>
<td>Cloud Content Contention</td>
</tr>
<tr>
<td>2011</td>
<td>Dhaval Vyas (UT)</td>
<td>Designing for Awareness: An Experience-focused HCI Perspective</td>
</tr>
<tr>
<td>2011</td>
<td>Carmen Bratosin (TUE)</td>
<td>Grid Architecture for Distributed Process Mining</td>
</tr>
<tr>
<td>2011</td>
<td>Xiaoyu Mao (UV)</td>
<td>Airport under Control. Multiagent Scheduling for Airport Ground Handling</td>
</tr>
<tr>
<td>2011</td>
<td>Milan Lovric (EUR)</td>
<td>Behavioral Finance and Agent-Based Artificial Markets</td>
</tr>
<tr>
<td>2011</td>
<td>Marijn Koolen (UV)</td>
<td>The Meaning of Structure: the Value of Link Evidence for Information Retrieval</td>
</tr>
<tr>
<td>2011</td>
<td>Maarten Schadd (UM)</td>
<td>Selective Search in Games of Different Complexity</td>
</tr>
<tr>
<td>2011</td>
<td>Jiyin He (UV)</td>
<td>Exploring Topic Structure: Coherence, Diversity and Relatedness</td>
</tr>
<tr>
<td>2011</td>
<td>Mark Ponsen (UM)</td>
<td>Strategic Decision-Making in complex games</td>
</tr>
<tr>
<td>2011</td>
<td>Ellen Rusman (OU)</td>
<td>The Mind’s Eye on Personal Profiles</td>
</tr>
<tr>
<td>2011</td>
<td>Qing Gu (UV)</td>
<td>Guiding service-oriented software engineering - A view-based approach</td>
</tr>
<tr>
<td>2011</td>
<td>Linda Terlouw (TUD)</td>
<td>Modularization and Specification of Service-Oriented Systems</td>
</tr>
<tr>
<td>2011</td>
<td>Junte Zhang (UV)</td>
<td>System Evaluation of Archival Description and Access</td>
</tr>
<tr>
<td>2011</td>
<td>Wouter Weekamp (UV)</td>
<td>Finding People and their Utterances in Social Media</td>
</tr>
<tr>
<td>2011</td>
<td>Herwin van Welbergen (UT)</td>
<td>Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scheduling and Realizing Multimodal Virtual Human Behavior</td>
</tr>
<tr>
<td>2011</td>
<td>Syed Waqar ul Goujain Jaffry (VU)</td>
<td>Analysis and Validation of Models for Trust Dynamics</td>
</tr>
<tr>
<td>2011</td>
<td>Matthias Aart Pontier (VU)</td>
<td>Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Trade-Offs in Embodied Conversational Agents and Robots</td>
</tr>
<tr>
<td>2011</td>
<td>Aniel Bhuilai (VU)</td>
<td>Dynamic website optimization through autonomous management of design patterns</td>
</tr>
<tr>
<td>2011</td>
<td>Rianne Kaptein(UVA)</td>
<td>Effective Focused Retrieval by Exploiting Query Context and Document Structure</td>
</tr>
<tr>
<td>2011</td>
<td>Faisal Kamiran (TUE)</td>
<td>Discrimination-aware Classification</td>
</tr>
<tr>
<td>2011</td>
<td>Egon van den Broek (UT)</td>
<td>Affective Signal Processing (ASP): Unraveling the mystery of emotions</td>
</tr>
<tr>
<td>2011</td>
<td>Ludo Waltman (EUR)</td>
<td>Computational and Game-Theoretic Approaches for Modeling Bounded Rationality</td>
</tr>
<tr>
<td>2011</td>
<td>Nees-Jan van Eck (EUR)</td>
<td>Methodological Advances in Bibliometric Mapping of Science</td>
</tr>
<tr>
<td>2011</td>
<td>Tom van der Weide (UU)</td>
<td>Arguing to Motivate Decisions</td>
</tr>
<tr>
<td>2011</td>
<td>Paolo Turini (UU)</td>
<td>Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations</td>
</tr>
<tr>
<td>2011</td>
<td>Maaike Harbers (UU)</td>
<td>Explaining Agent Behavior in Virtual Training</td>
</tr>
<tr>
<td>2011</td>
<td>Erik van der Spek (UU)</td>
<td>Experiments in serious game design: a cognitive approach</td>
</tr>
<tr>
<td>2011</td>
<td>Adriana Burlutiu (RUN)</td>
<td>Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Network Inference</td>
</tr>
<tr>
<td>2011</td>
<td>Nyree Lemmens (UM)</td>
<td>Bee-inspired Distributed Optimization</td>
</tr>
<tr>
<td>2011</td>
<td>Joost Westra (UU)</td>
<td>Organizing Adaptation using Agents in Serious Games</td>
</tr>
<tr>
<td>2011</td>
<td>Viktor Clerc (VU)</td>
<td>Architectural Knowledge Management in Global Software Development</td>
</tr>
<tr>
<td>2011</td>
<td>Luan Ibraim (UT)</td>
<td>Cryptographically Enforced Distributed Data Access Control</td>
</tr>
<tr>
<td>2011</td>
<td>Michal Sindiar (UU)</td>
<td>Explaining Behavior through Mental State Attribution</td>
</tr>
<tr>
<td>2011</td>
<td>Henk van der Schuur (UU)</td>
<td>Process Improvement through Software Operation Knowledge</td>
</tr>
<tr>
<td>2011</td>
<td>Boris Reuderink (UT)</td>
<td>Robust Brain-Computer Interfaces</td>
</tr>
<tr>
<td>2011</td>
<td>Herman Stehouwer (UV)</td>
<td>Statistical Language Models for Alternative Sequence Selection</td>
</tr>
<tr>
<td>2011</td>
<td>Beihe Li (TUD)</td>
<td>Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work</td>
</tr>
<tr>
<td>2011</td>
<td>Azizi Bin Ab Aziz(VU)</td>
<td>Exploring Computational Models for Intelligent Support of Persons with Depression</td>
</tr>
<tr>
<td>2011</td>
<td>Mark Ter Maat (UT)</td>
<td>Response Selection and Turn-taking for a Sensitive Artificial Listening Agent</td>
</tr>
<tr>
<td>2011</td>
<td>Andreea Niculescu (UT)</td>
<td>Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality</td>
</tr>
</tbody>
</table>

2012

2012-01 Terry Kaketo (UV) | Relationship Marketing for SMEs in Uganda

2012-03 Adam Vanya (VU) | Supporting Architecture Evolution by Mining Software Repositories

2012-04 Jurriaan Souer (UU) | Development of Content Management System-based Web Applications

2012-05 Marijn Plomp (UU) | Maturing Interorganisational Information Systems
2012-06 Wolfgang Reinhardt (OU) Awareness Support for Knowledge Workers in Research Networks
2012-07 Rianne van Lambalgen (VU) When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions
2012-08 Gerben de Vries (UvA) Kernel Methods for Vessel Trajectories
2012-09 Ricardo Neisse (UT) Trust and Privacy Management Support for Context-Aware Service Platforms
2012-10 David Smits (TUE) Towards a Generic Distributed Adaptive Hypermedia Environment
2012-11 J.C.B. Rantham Prabhakara (TUE) Process Mining in the Large: Preprocessing, Discovery, and Diagnostics
2012-12 Kees van der Sluijs (TUE) Model Driven Design and Data Integration in Semantic Web Information Systems
2012-13 Suleman Shahid (UvT) Fun and Face: Exploring non-verbal expressions of emotion during playful interactions
2012-14 Evgeny Knutov (TUE) Generic Adaptation Framework for Unifying Adaptive Web-based Systems
2012-15 Natalie van der Wal (VU) Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes
2012-16 Fiemke Both (VU) Helping people by understanding them - Ambient Agents supporting task execution and depression treatment
2012-17 Amal Elgammal (UvT) Towards a Comprehensive Framework for Business Process Compliance
2012-18 Eltjo Poort (VU) Improving Solution Architecting Practices
2012-19 Helen Schonenberg (TUE) What’s Next? Operational Support for Business Process Execution
2012-20 Ali Bahramisharif (RUN) Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing
2012-21 Roberto Comacchia (TUD) Querying Sparse Matrices for Information Retrieval
2012-22 Thijs Vis (UvT) Intelligence, politie en veiligheidsdienst: verenigbare grootheden?
2012-23 Christian Muehl (UT) Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction
Appendix I. \( \Psi \) Processor Simulation, Verification and Validation

Abstract. This Appendix addresses the necessary verification process of the \( \Psi \) processor specifications; are the \( \Psi \) processor specifications of section 4 and 5 formal correct and comply with the axioms of the \( \Psi \) theory that are the foundation of the axiomatic specifications? Verification of the \( \Psi \) processor specifications 'on paper' is prohibitively difficult, just as error-prone as the derivation and delivers no reliable results. A better approach is the implementation in software, the \( \Psi \) processor, and verification of the functional behavior of the \( \Psi \) processor against the \( \Psi \) theory axioms. In addition there is a validation testing cycle, specific phenomena, especially rollback phenomena discussed in section 6.2, have to be investigated.

Introduction

The \( \Psi \) processor calculates transaction state transitions for aggregated \( \Psi \) models that are composed of an unlimited number of transactions. The state transitions of such a model are calculated using the set of state transaction functions of section 5. The transaction axiom (section 2.5) states that any transaction between actors that occurs in the world of phenomena follows a specific pattern and there are three of these patterns, the basic pattern, the standard pattern and the pattern with cancellations (section 5.6).

The verification\(^{107}\) process verifies that for any transaction, in any aggregated demo model under execution, for any transaction state, that any state transition must comply precisely with the transaction axiom (three patterns), without lack of anomalies.

\(^{107}\) Verification refers to “internal” correctness of the model. Validation refers to the degree that the behavior of the model complies with phenomena we observe in enterprises, e.g. the compliance to the transaction axiom.
The validation process validates that any phenomenon observed in reality complies with the behavior of the demo model under execution.

The following topics have to be investigated.

i.) The question is if such a set of state transition functions exists at all. The answer is that such a set must exist. We see empirically that in the world of phenomena, social human systems with nested transactions exist and all transactions are executed, which results in a final production. We may find two (or more) sets of state transition functions. If the verification for both sets of functions is correct, we conclude that these sets of functions are equivalent.

ii.) The question is what the set of state transition functions is, which is answered in section 5.

iii.) The question is if the set of state transitions is correct, complete, without redundancy, without ambiguity, without deadlocks and without anomalies. The verification of correctness involves in the first place that the verification for each transaction of a nested transaction system complies completely with the transaction axiom for this transaction. The completeness requirement involves the verification that each state transition specified in the transaction axiom is implemented in the set of functions. A missing state transition function might result in a deadlock situation. Redundancy refers to the existence of two or more identical state transition functions, which is easy to identify. Ambiguity refers to the situation that for a specific conditions there are more than one state transitions possible, which is also easy to identify. Deadlock conditions may occur if two processes are waiting for each other. Anomalies refer to any other conditions that should not occur.

iv.) The question is if the validation of the behavior of aggregated \( \Psi \) models against certain phenomena observed in the world of reality is correct. This involves validation of the upward and downward transaction rollback phenomena and cancellation patterns.

### Appendix I.1 Verification and Validation Procedures

The following remarks can be made about testing.

The \( \Psi \) processor has been subjected to 'traditional' pragmatic software testing methods. The number of tests is limited, there is no completeness in testing. Each verification step has also 'manually' tested; not against an external software engine, which is much less error-prone. Deadlock conditions should not occur in the \( \Psi \) processor. They may however occur in any DEMO processor model because deadlocks expressed in DEMO Action Model rules may exist in the real world also.

The level of testing in this Appendix delivers a pragmatic level of confidence, but no formal certainty. For these kinds of software systems, the formal quality is mandatory.
Monte Carlo Simulations
A better approach with a higher degree of completeness and confidence is the Monte Carlo method. The $\Psi$ processor is tested against an 'inverse $\Psi$ processor' that verifies any anomalies. The tests are driven and configured by a random number generator. There is a higher level confidence after a high number of tests, but without a claim for the mandatory formal completeness.

*Old Monte Carloists never die, they run out of random numbers.*

- anonymous gamblers and software engineers -

Temporal Logic Simulations
The completeness requirement of the verification against a large number of different $\Psi$ models is a problem outside the scope of this research and cannot be done 'manually' as in this Appendix. This requirement may be addressed with success if temporal logic verification (section 7.4.1) is applied in a proper way.

Extensive independent validation and model compliance tests have been carried out manually also by Guerreiro [Guerreiro S, et all, 2011, 2011]. Runtime transactions and access control modes in enterprise simulation have been validated. No anomalies were found.

The validations and verifications have been made using the $\Psi$ processor executing a model. All actions and state transitions are logged to an external logging text file for verification. The logfile is generated, but at some places, textual information has been added.

Appendix I.2 The elementary $\Psi$ Model Validation
The elementary $\Psi$ model, shown in fig. 8.1 has to be validated against the transaction axiom first and in full detail.

![Figure 8.1. The elementary $\Psi$ model.](image)
Appendix I. Ψ Processor Simulation, Verification and Validation

The Ψ model of fig 8.1 rendered by the Ψ processor in DMOL is represented in this file (after white space formatting for better overview):

```xml
<?xml version="1.0" encoding="windows-1252" ?>
<DMApp>
  <DEMOApplication>
    <PfactChilds>
      <Pfact>
        <Pfact-T00>
          <PfactPublic>
            prmPfactIdentifier = "Pfact-T00"
            prmPfactLabel = ""
          </PfactPublic>
        </Pfact>
        <CFact>
          <CFactPrivate>
            prmRequested = "false"  prmPromised = "false"  prmStated = "false"
            prmAccepted = "false"
            prmDeclined = "false"  prmQuited = "false"  prmRejected = "false"
            prmStopped = "false"
          </CFactPrivate>
          <CFactPublic>
            prmCFactIdentifier = "Cfact-T00"
            prmCFactLabel = ""
          </CFactPublic>
        </CFact>
      </Pfact>
    </PfactChilds>
  </DEMOApplication>
</DMApp>
```
This file can be rendered and parsed again by the $\Psi$ processor to reconstruct the original executable model in its correct state. Note that this representation differs slightly from the grammar specified in section 4.2; it is extended but still complies with the grammar. For each component ( Pfact, Cfact, Actor ) a separation has been made a public part and a private part to comply with ‘clean’ object oriented programming principles. Also a unique identifier and a label, some textual description, have been declared. While some objects may have empty public or private parts this is probably not the case in more extended implementations of the DEMO processor.

The representation as NLDMOL is:

```
DEMO model 'DEMOApplication' 'Test version DEMO Tree'.

Declaration of Actors.
Root Actor-Initiator 'A00' 'Root Initiator';
Actor 'A01';

Declaration of Transactions.
Transaction 'Pfact-T00';

Declaration of Transaction aggregation.
Actor-Initiator 'A00' 'Root Initiator'
  requires from:
    Actor-Executor 'A01' performing 'Pfact-T00'

Declaration of Transaction states.
Actor-Initiator 'A00' 'Root Initiator' for Transaction 'Pfact-T00' has
Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A01'.
Query Actor-Initiator 'Request Yes|No'.
```

The following lines specify the Cfact state:

```
<CFactPrivate prmRequested="false" prmPromised="false" prmStated="false"
  prmAccepted="false" prmDeclined="false" prmQuited="false"
  prmRejected="false" prmStopped="false">
  </CFactPrivate>
```

which is state Cfact S0:
Appendix I. \( \Psi \) Processor Simulation, Verification and Validation

\[ \begin{cases} R_q, P_m, D_c, Q_t = o \\ S_t, A_c, R_j, S_p = o \end{cases} \]

Cfact S0 is the default state.

The \( \Psi \) processor executes the state transitions and logs which axioms have been applied.

Simulation I.
The simulation of the standard transaction pattern, Request – Promise – State – Accept results in the following logging. First the Request – promise is shown:

[Axiom1] applied; of Pfact 'Pfact-T00, ', A00, Root Initiator'

Actor 'A00, Root Initiator' with role 'Root Initiator' of Pfact 'Pfact-T00, '. executing [Axiom1] selected option 'Yes' of 'requests 'Yes | No''.

[Axiom5] applied; of Pfact 'Pfact-T00, ', A01, '

Actor 'A01, ' with role 'Executor'; of Pfact 'Pfact-T00, '. executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

First [Axiom1] has been applied and the Root Initiator selected option ‘Yes’ of options ‘Yes|No’ resulting in Request being issued.

[Axiom5] has been applied, the Executor selected the option ‘Promise’ of the two options ‘Promise|Decline’ resulting in Promise being issued and Request being reset.

Rendering the model in DMOL at this stage delivers this file:

```xml
<?xml version="1.0" encoding="windows-1252" ?>
<DMApp>DEMOApplication
  <PfactChilds>
    <Pfact>Pfact-T00
      <PfactPublic prmPfactIdentifier="Pfact-T00" prmPfactLabel="">
      </PfactPublic>
      </Pfact>
      <CFact>
        <CFactPrivate prmRequested="false" prmPromised="true"
                        prmStated="false" prmAccepted="false" prmDeclined="false" prmQuited="false"
                        prmRejected="false" prmStopped="false">
        </CFactPrivate>
        <CFactPublic prmCfactIdentifier="Cfact-T00" prmCfactLabel="">
        </CFactPublic>
    </Pfact>
  </PfactChilds>
</DMApp>
```
The value of Promised is now true. The value of Requested was true and has been reset. The execution is continued resulting in this logging:

[Axiom21] applied; of Pfact 'Pfact-T00, '., A01, '

Actor 'A01, ' with role 'Executor'; of Pfact 'Pfact-T00, '. executing [Axiom21] selected option 'State' of 'Executed State'.

[Axiom24] applied; Pfact-T00

Actor 'A00, Root Initiator' with role 'ROOT Initiator'; of Pfact 'Pfact-T00, '. executing [Axiom24-25] selected option 'Accept' of 'Accept | Reject'.

Rendering the $\Psi$ model now yields:

<?xml version="1.0" encoding="windows-1252" ?>
<DMApp>DEMOApplication
<PfactChilds>
<Pfact>Pfact-T00
<PfactPublic prmPfactIdentifier="Pfact-T00" prmPfactLabel=""
</PfactPublic>
</Pfact>
</PfactChilds>
</DMApp>
Appendix I. Processor Simulation, Verification and Validation

The Cfact state is now:

\[
\begin{array}{c}
\text{Cfact} \text{ S11 states that the Initiator issued an Accept, the} \\
\text{transaction has completed.} \\
\text{At this stage the transaction has completed, the production Pfact-T00 has been accepted.}
\end{array}
\]
Simulation II.
The simulation is now repeated for a different situation where minor loops occur:
1. The Request is followed by a Decline, a new Request and a Promise.
2. The State is followed by a Reject, a new State and an Accept.

This simulation delivers the following logging with the Cfact states inserted:

[Cfact S0]
\[
\begin{pmatrix}
\text{Rq}_0, \text{Pr}_0, \text{Dc}_0, \text{Qt} = o \\
\text{St}_1, \text{Ac}_1, \text{Rj}_1, \text{Sp} = o
\end{pmatrix}
\]

Cfact S0 is the default state, prior to the Transaction start.

[Axiom1] applied; of Pfact 'Pfact-T00, \', A00, Root Initiator'

Actor 'A00, Root Initiator' with role 'Root Initiator' of Pfact 'Pfact-T00, \'.
exercing [Axiom1] selected option 'Yes' of 'requests 'Yes | No''.

[Cfact S1]
\[
\begin{pmatrix}
\text{Rq} = 1 \\
\text{Pr}, \text{Dc}_0, \text{Qt} = o \\
\text{St}_1, \text{Ac}_1, \text{Rj}_1, \text{Sp} = o
\end{pmatrix}
\]

Cfact S1, a Request has been issued by the Initiator.

[Axiom6] applied; of Pfact 'Pfact-T00, \', A01, '

Actor 'A01, ' with role 'Executor'; of Pfact 'Pfact-T00, '\n'exercing [Axiom5][Axiom6] selected option 'Decline' of 'Promise | Decline'.

[Cfact S2]
\[
\begin{pmatrix}
\text{Rq} = 1, \text{Pr} = o \\
\text{Dc} = 1 \\
\text{Qt} = 0 \\
\text{St}_1, \text{Ac}_1, \text{Rj}_1, \text{Sp} = o
\end{pmatrix}
\]

Cfact S2, a Decline has been issued by the Executor.

[Axiom7] applied, of Pfact 'Pfact-T00, \', A00, Root Initiator'

Actor 'A00, Root Initiator' with role 'Initiator' of Pfact 'Pfact-T00, '
'exercising [Axiom7-8-9-10-11-12-13-a] selected option 'Request' of 'Request| Quit'.
Appendix I. Ψ Processor Simulation, Verification and Validation

[Cfact S1]
\[ R_q = 1; P_m, D_c, Q_t = 0 \]
Cfact S1, a Request has been issued by the Initiator.

[Axiom5] applied; of Pfact 'Pfact-T00, ., A01, '
Actor 'A01, ' with role 'Executor'; of Pfact 'Pfact-T00, '.
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

[Cfact S3]
\[ R_q = 0; P_m = 1; D_c, Q_t = 0; S_t, A_c, R_j, S_p = 0 \]
Cfact S3, a Promise has been issued by the Executor.

[Axiom21] applied; of Pfact 'Pfact-T00, ., A01, '
Actor 'A01, ' with role 'Executor'; of Pfact 'Pfact-T00, '.
executing [Axiom21] selected option 'State' of 'Executed State'.

[Cfact S6]
\[ R_q = 0; P_m = 1; D_c, Q_t = 0; S_t = 1; A_c, R_j, S_p = 0 \]
Cfact S6, a State has been issued by the Executor.

[Axiom25] applied; Pfact-T00
Actor 'A00, Root Initiator' with role 'ROOT Initiator'; of Pfact 'Pfact-T00, '.
executing [Axiom24-25] selected option 'Reject' of 'Accept | Reject'.


Appendix I. \( \Psi \) Processor Simulation, Verification and Validation

\[ \text{Cfact S7} \]
\[
\begin{align*}
Rq & = 0 \\
Pm & = 1 \\
Dc Qt & = 0 \\
St & = 1 \\
Ac & = 0 \\
Rj & = 1 \\
Sp & = 0
\end{align*}
\]

\textbf{Cfact S7} states that the Initiator issued a Reject, after a State from the Executor.

[Axiom26] applied; of Pfact 'Pfact-T00, \', A01, \'
Actor 'A01, ' with role 'Executor'; of Pfact 'Pfact-T00, \'.
executing [Axiom26-27] selected option 'State' of 'State | Stop'.

\[ \text{Cfact S6} \]
\[
\begin{align*}
Rq & = 0 \\
Pm & = 1 \\
Dc Qt & = 0 \\
St & = 1 \\
Ac, Rj, Sp & = 0
\end{align*}
\]

\textbf{Cfact S6} states that a State has been issued by the Executor.

[Axiom24] applied; Pfact-T00
Actor 'A00, Root Initiator' with role 'ROOT Initiator'; of Pfact 'Pfact-T00, \'.
executing [Axiom24-25] selected option 'Accept' of 'Accept | Reject'.

\[ \text{Cfact S11} \]
\[
\begin{align*}
Rq, Pm, Dc Qt & = 0 \\
St & = 0 \\
Ac & = 1 \\
Rj, Sp & = 0
\end{align*}
\]

\textbf{Cfact S11} states that the Initiator issued an Accept, the transaction has completed.
Simulation III.
Now the simulation for this model is repeated with a Request – Promise – State - Reject - Stop, resulting in this logging with inserted Cfact states.

[Cfact S0]
\[
\begin{align*}
R_q, P_m, D_c, O_t & = 0 \\
S_t, A_c, R_j, S_p & = 0
\end{align*}
\]

Cfact S0 is the default state, prior to the transaction start.

[Axiom1] applied; of Pfact 'Pfact-T00, ., A00, Root Initiator'

Actor 'A00, Root Initiator' with role 'Root Initiator' of Pfact 'Pfact-T00, .',
executing [Axiom1] selected option 'Yes' of 'requests 'Yes | No''.

[Cfact S1]
\[
\begin{align*}
R_q & = 1 \\
P_m, D_c, O_t & = 0 \\
S_t, A_c, R_j, S_p & = 0
\end{align*}
\]

Cfact S1 states that a Request has been issued by the Initiator.

[Axiom5] applied; of Pfact 'Pfact-T00, ., A01, '

Actor 'A01, ' with role 'Executor'; of Pfact 'Pfact-T00, .',
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

[Cfact S3]
\[
\begin{align*}
R_q & = 0 \\
P_m & = 1 \\
D_c, O_t & = 0 \\
S_t, A_c, R_j, S_p & = 0
\end{align*}
\]

Cfact S3 states that a Promise has been issued by the Executor.

[Axiom21] applied; of Pfact 'Pfact-T00, ., A01, '

Actor 'A01, ' with role 'Executor'; of Pfact 'Pfact-T00, .',
execting [Axiom21] selected option 'State' of 'Executed State'.
Appendix I. \( \Psi \) Processor Simulation, Verification and Validation

[Cfact S6]
\[
\begin{align*}
Rq &= 0 \\
Prm &= 1 \\
Dc,Qt &= 0 \\
St &= 1 \\
Ac,Rj,Sp &= 0
\end{align*}
\]

Cfact S6 states that a State has been issued by the Executor.

[Axiom25] applied; Pfact-T00

Actor 'A00, Root Initiator' with role 'ROOT Initiator'; of Pfact 'Pfact-T00,' executing [Axiom24-25] selected option 'Reject' of 'Accept | Reject'.

[Cfact S7]
\[
\begin{align*}
Rq &= 0 \\
Prm &= 1 \\
Dc,Qt &= 0 \\
St &= 1 \\
Ac &= 0 \\
Rj &= 1 \\
Sp &= 0
\end{align*}
\]

Cfact S7, Initiator issued a Reject, after a State from the Executor.

[Axiom27] applied; of Pfact 'Pfact-T00, '., A01,

Actor 'A01, ' with role 'Executor'; of Pfact 'Pfact-T00, ' executing [Axiom26-27] selected option 'Stop' of 'State | Stop'.

[Cfact S9]
\[
\begin{align*}
Rq,Prm,Dc,Qt &= 0 \\
St,Ac,Rj &= 0 \\
Sp &= 1
\end{align*}
\]

Cfact S9, Executor issued a Stop, after a Reject from the Initiator.

[Axiom28] applied; Pfact-T00

Actor 'A00, Root Initiator' with role 'ROOT Initiator'; of Pfact 'Pfact-T00, '

executing [Axiom28-29-30-31-32-33] selected option 'Acknowledge' of 'Stop - Acknowledge'.

Appendix I. \( \Psi \) Processor Simulation, Verification and Validation

[Cfact S0]
\[
\begin{align*}
Rq, Pm, Dc, Qt &= 0 \\
St, Ac, Rj, Sp &= 0
\end{align*}
\]
[Cfact S0] is the default state, prior to the transaction start.

Note that the Stop – Acknowledge [Axiom28], resets the Stop. The model is now in the default state S0.

Simulation IV.
The following simulation shows a Request – Decline – Quit – Quit-Acknowledge cycle.

[Cfact S0]
\[
\begin{align*}
Rq, Pm, Dc, Qt &= 0 \\
St, Ac, Rj, Sp &= 0
\end{align*}
\]
[Cfact S0] is the default state, prior to the transaction start.

[Axiom1] applied; of Pfact 'Pfact-T00, '.' A00, Root Initiator'
Actor 'A00, Root Initiator' with role 'Root Initiator' of Pfact 'Pfact-T00,'.
executing [Axiom1] selected option 'Yes' of 'requests 'Yes | No''.

[Cfact S1]
\[
\begin{align*}
Rq &= 1 \\
Pm, Dc, Qt &= 0 \\
St, Ac, Rj, Sp &= 0
\end{align*}
\]
[Cfact S1] states that a Request has been issued by the Initiator.

[Axiom6] applied; of Pfact 'Pfact-T00, '.' A01, '
Actor 'A01, ' with role 'Executor'; of Pfact 'Pfact-T00,'.
executing [Axiom5][Axiom6] selected option 'Decline' of 'Promise | Decline'.

[Cfact S2]
\[
\begin{align*}
Rq, Pm &= 0 \\
Dc &= 0 \\
Qt &= 0 \\
St, Ac, Rj, Sp &= 0
\end{align*}
\]
[Cfact S2] states that a Decline has been issued by the Executor.

[Axiom9] applied; Pfact-T00
Appendix I. $\Psi$ Processor Simulation, Verification and Validation

Actor 'A00, Root Initiator' with role 'Initiator' of Pfact 'Pfact-T00, '. executing [Axiom7-8-9-10-11-12-13-a] selected option 'Quit' of 'Request|Quit'.

[Cfact S5]

\[
\begin{align*}
Rq,Prm &= o \\
Dc,Qt &= 1 \\
S_t,Ac,Rj,Sp &= o
\end{align*}
\]

\textbf{Cfact S5} states that a Quit has been issued by the Initiator during a pending Decline.

[Axiom14] applied; of Pfact 'Pfact-T00, ', A01, '

Actor 'A01, ' with role 'Executor'; of Pfact 'Pfact-T00, '. executing Axiom14-15-16-17-18-19] selected option 'Acknowledge' of 'Quit - Acknowledge'.

[Cfact S0]

\[
\begin{align*}
Rq,Prm,Dc,Qt &= o \\
S_t,Ac,Rj,Sp &= o
\end{align*}
\]

\textbf{Cfact S0} is the default state, prior to the transaction start.

Verification of Cfact states S0..S11 with rendered DMOL Cfact states proves correctness of the allowed states.

Conclusions of the validation of the elementary $\Psi$ model are that validation of the elementary model is proven correct, compliant with the transaction axiom, and that no anomalies have been found.

**Appendix I.3 Validation of a concatenated $\Psi$ Model**

In this section, an aggregated $\Psi$ model is investigated as example and some phenomena described in section 6 are shown. In this example, a chain of three nested or concatenated transactions is investigated.

![Figure 8.2](image.png)

\textbf{Figure 8.2.} Model composed of three nested transactions.
The transactions Txx are each composed of a Pfact and a Cfact, declared as “Pfact-Txx” and “Cfact-Txx” in the $\Psi$ model.

The NLDMOL Aspect model of fig. 8.2 becomes:

DEM0 model 'DEM0 Application' 'Test version DEM0 Tree'.

Declaration of Actors.
Root Actor-Initiator 'A00' 'Root Initiator';
Actor 'A01';
Actor 'A10';
Actor 'A20';

Declaration of Transactions.
Transaction 'Pfact-T00';
Transaction 'Pfact-T10';
Transaction 'Pfact-T20';

Declaration of Transaction aggregation.
Actor-Initiator 'A00' 'Root Initiator'
    requires from:
    Actor-Executor 'A01' performing 'Pfact-T00'
    requires from:
    Actor-Executor 'A10' performing 'Pfact-T10'
    requires from:
    Actor-Executor 'A20' performing 'Pfact-T20'
end of require;
end of require;
end of require.

Declaration of Transaction states.
Actor-Initiator 'A00' 'Root Initiator' for Transaction 'Pfact-T00' has Cfact state of:
    Requested = false; Promised = false; Declined = false; Quited = false;
    Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A01'.
Query Actor-Initiator 'Request Yes|No'.

Actor-Initiator 'A01' for Transaction 'Pfact-T10' has Cfact state of:
    Requested = false; Promised = false; Declined = false; Quited = false;
    Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A10'.

Actor-Initiator 'A10' for Transaction 'Pfact-T20' has Cfact state of:
    Requested = false; Promised = false; Declined = false; Quited = false;
    Stated = false; Rejected = false; Stopped = false; Accepted = false;
Appendix I. Ψ Processor Simulation, Verification and Validation

with Actor-Executor 'A20'.

The model of fig. 8.2 in DMOL representation (with white space formatting):

```xml
<?xml version="1.0" encoding="windows-1252" ?>
<DMApp>DEMOApplication
<PfactChilds>
  <Pfact>Pfact-T00
    <PfactPublic prmPfactIdentifier="Pfact-T00" prmPfactLabel=""/>
  </PfactPublic>
  <CFact>
    <CFactPrivate prmRequested="false" prmPromised="false"
    prmStated="false" prmAccepted="false" prmDeclined="false" prmQuited="false"
    prmRejected="false" prmStopped="false">
    <CFactPublic prmCFactIdentifier="Cfact-T00" prmCFactLabel=""/>
    </CFactPublic>
  </CFact>
  <ActExec>
    <Act>
      <ActPublic prmActIdentifier="A01" prmActLabel=""/>
    </ActPublic>
  </ActExec>
  <ActInit>
    <Act>
      <ActPublic prmActIdentifier="A00" prmActLabel="Root Initiator">
    </ActPublic>
  </ActInit>
  </PfactChilds>
  <Pfact>Pfact-T10
    <PfactPublic prmPfactIdentifier="Pfact-T10" prmPfactLabel=""/>
  </PfactPublic>
  <CFact>
    <CFactPrivate prmRequested="false" prmPromised="false"
    prmStated="false" prmAccepted="false" prmDeclined="false" prmQuited="false"
    prmRejected="false" prmStopped="false">
    <CFactPublic prmCFactIdentifier="Cfact-T10" prmCFactLabel=""/>
    </CFactPublic>
  </CFact>
</CollApp>
```
Simulation V
The purpose of this simulation is to validate the upward propagation of the Quit-QuitAcknowledge cycle. The following simulation is carried out:
The transaction starts with a Request to $T_0$; a Promise of $T_0$; a Request for $T_0$; a Promise for $T_0$; a Request for $T_0$; a Promise for $T_0$; a State for $T_0$; a Reject for $T_0$; a Stop for $T_0$; a Stop Acknowledge for $T_0$; a Quit for $T_0$, a Quit Acknowledge for $T_0$; a Quit for $T_0$;

The $\Psi$ processor simulation yields this logging, with the appropriate Cfact states and some /* comments */ are inserted:

[Cfact $S_0$]
$$\begin{align*}
R_q, P_m, D_c, Q_r & = 0 \\
St, Ac, R_j, S_p & = 0
\end{align*}$$

Cfact $S_0$, default state for Cfacts $T_0$, $T_0$ and $T_0$.

[Axiom1] applied; of Pfact 'Pfact-T00, ', $A_0$, Root Initiator'
Actor 'A00, Root Initiator' with role 'Root Initiator' of Pfact 'Pfact-T00, '.
executing [Axiom1] selected option 'Yes' of 'requests 'Yes | No''.

/* From this point the downward propagation of Request – Promise – Request – Promise etc is shown. */

[Cfact $S_1$]
$$\begin{align*}
R_q & = 1 \\
P_m, D_c, Q_r & = 0 \\
St, Ac, R_j, S_p & = 0
\end{align*}$$

Cfact $S_1$, a Request issued by the Initiator $CA_0$ for $T_0$.

[Axiom5] applied; of Pfact 'Pfact-T00, ', $A_1$, '
Actor 'A01, ' with role 'Executor'; of Pfact 'Pfact-T00, '.
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

[Cfact $S_3$]
$$\begin{align*}
R_q & = 0 \\
P_m & = 1 \\
D_c, Q_r & = 0 \\
St, Ac, R_j, S_p & = 0
\end{align*}$$

Cfact $S_3$, a Promise issued by the Executor $CA_1$ for $T_0$.

[Axiom3] applied; Pfact-T10
Appendix I. Ψ Processor Simulation, Verification and Validation

[Cfact S1]
\[
\begin{array}{l}
Rq = 1 \\
Pm, Dc, Qt = 0 \\
St, Ac, Rj, Sp = 0
\end{array}
\]

Cfact S1, a Request issued by the Initiator CA01 for T10.

[Axiom5] applied; of Pfact 'Pfact-T10, '., A10, '

Actor 'A10, ' with role 'Executor'; of Pfact 'Pfact-T10, '.
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

[Cfact S3]
\[
\begin{array}{l}
Rq = 0 \\
Pm = 1 \\
Dc, Qt = 0 \\
St, Ac, Rj, Sp = 0
\end{array}
\]

Cfact S3, a Promise issued by the Executor CA10 for T10.

[Axiom3] applied; Pfact-T20

[Cfact S1]
\[
\begin{array}{l}
Rq = 1 \\
Pm, Dc, Qt = 0 \\
St, Ac, Rj, Sp = 0
\end{array}
\]

Cfact S1, a Request issued by the Initiator CA10 for T20.

[Axiom5] applied; of Pfact 'Pfact-T20, '., A20, '

Actor 'A20, ' with role 'Executor'; of Pfact 'Pfact-T20, '.
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

[Cfact S3]
\[
\begin{array}{l}
Rq = 0 \\
Pm = 1 \\
Dc, Qt = 0 \\
St, Ac, Rj, Sp = 0
\end{array}
\]

Cfact S3, a Promise issued by the Executor CA20 for T20.

[Axiom21] applied; of Pfact 'Pfact-T20, '., A20, '

Actor 'A20, ' with role 'Executor'; of Pfact 'Pfact-T20, '.
executing [Axiom21] selected option 'State' of 'Executed State'.


Appendix I. Ψ Processor Simulation, Verification and Validation

[Cfact S6]
\[
\begin{align*}
Rq &= 0 \\
Prm &= 1 \\
Dc,Qt &= 0 \\
St &= 1 \\
Ac, Rj, Sp &= 0
\end{align*}
\]

Cfact S6, a State issued by the Executor CA20 for T20.

[Axiom25] applied; Pfact-T20

Actor 'A10,' with role 'Initiator'; of Pfact 'Pfact-T20,' executing [Axiom24-25] selected option 'Reject' of 'Accept | Reject'.

[Cfact S7]
\[
\begin{align*}
Rq &= 0 \\
Prm &= 1 \\
Dc,Qt &= 0 \\
St &= 1 \\
Ac &= 0 \\
Rj &= 1 \\
Sp &= 0
\end{align*}
\]

Cfact S7, the Initiator CA10 issued a Reject for T20.

[Axiom27] applied; of Pfact 'Pfact-T20,'., A20,' Actor 'A20,' with role 'Executor'; of Pfact 'Pfact-T20,' executing [Axiom26-27] selected option 'Stop' of 'State | Stop'.

[Cfact S9]
\[
\begin{align*}
Rq, Prm, Dc, Qt &= 0 \\
St, Ac, Rj &= 0 \\
Sp &= 1
\end{align*}
\]

Cfact S9, the Executor CA20 issued a Stop for T20.

[Axiom29] applied; Pfact-T10

[Cfact S2]
\[
\begin{align*}
Rq, Prm &= 0 \\
Dc &= 1 \\
Qt &= 0 \\
St, Ac, Rj, Sp &= 0
\end{align*}
\]

Cfact S2, a Decline issued by Executor CA10 for T10.
/* This is the beginning of the upward Decline-Quit cycle. */

[Axiom28] applied; Pfact-T20

Actor 'A10,' with role 'Initiator'; of Pfact 'Pfact-T20, '.
executing [Axiom28-29-30-31-32-33] selected option 'Acknowledge' of 'Stop Acknowledge'.

/* This logging of execution of one or more of these Axioms 29-33 and some others is rendered 'late', after logging of execution of Axiom29 and Axiom28. This is implementation dependent. */

[Cfact S0]
\[
\begin{align*}
Rq,Pm, & \hspace{1em} \text{default state, restored by CA10 for T20.} \\
(Tc,Qt = o) & \\
St, & \\
Ac,Rj,Sp = o
\end{align*}
\]

[Axiom8] applied; of Pfact 'Pfact-T10, '., A01, '

[Cfact S2]
\[
\begin{align*}
Rq,Pm & = o \\
Dc & = 1 \\
Qt & = o \\
St,Ac,Rj,Sp & = o
\end{align*}
\]

[Axiom9] applied; Pfact-T10

[Cfact S5]
\[
\begin{align*}
Rq,Pm & = o \\
Dc,Qt & = 1 \\
St,Ac,Rj,Sp & = o
\end{align*}
\]

Actor 'A01,' with role 'Initiator'; of Pfact 'Pfact-T10, '.
executing [Axiom7-8-9-10-11-12-13] selected option 'Quit' of 'Request|Quit'.

/* This logging of execution of one or more of these Axioms 7-13 is rendered 'late', after logging of execution of Axiom8 and Axiom9 */

[Axiom14] applied; of Pfact 'Pfact-T10, '., A10, '
Appendix I. Processor Simulation, Verification and Validation

Actor 'A10, ' with role 'Executor'; of Pfact 'Pfact-T10, '.
executing Axiom14-15-16-17-18-19] selected option 'Acknowledge' of 'Quit - Acknowledge'.

\[Cfact S0\]
\[
\begin{align*}
Rq, Pm, Dc, Qt & = 0 \\
St, Ac, Rj, Sp & = 0
\end{align*}
\]
Cfact S0, the default state, restored by CA10 for T10, after Quit-acknowledge.

[Axiom9] applied; Pfact-T00

Actor 'A00, Root Initiator' with role 'Initiator' of Pfact 'Pfact-T00, '.
exchanging [Axiom7-8-9-10-11-12-13-a] selected option 'Quit' of 'Request| Quit'.

\[Cfact S5\]
\[
\begin{align*}
Rq, Pm & = 0 \\
Dc, Qt & = 1 \\
St, Ac, Rj, Sp & = 0
\end{align*}
\]
Cfact S5, a Quit issued by Initiator CA00, during a pending Decline, for T00.

[Axiom14] applied; of Pfact 'Pfact-T00, ', A01, '

Actor 'A01, ' with role 'Executor'; of Pfact 'Pfact-T00, '.
execting Axiom14-15-16-17-18-19] selected option 'Acknowledge' of 'Quit - Acknowledge'.

\[Cfact S0\]
\[
\begin{align*}
Rq, Pm, Dc, Qt & = 0 \\
St, Ac, Rj, Sp & = 0
\end{align*}
\]
Cfact S0, default state, restored by CA01 for T00, after Quit-acknowledge.

The model, each transaction, is now again in its default state Cfact S0.
The following states have been reached: Cfact S0, S1, S2, S3, S5, S6, S7, S9.

Simulation VI
The simulation of the model of fig. 8.2 is now carried out for the situation where the transactions are “successful”, the so-called happy-flow, resulting in a final delivery of the production of T00.
Appendix I. $\Psi$ Processor Simulation, Verification and Validation

[Cfact S0]
\[
\begin{align*}
&Rq_{T01} \equiv 0, \\
&Pm_{T01} \equiv 0, \\
&St_{T01} \equiv 0, \\
&Ac_{T01} \equiv 0, \\
&Rj_{T01} \equiv 0, \\
&Sp_{T01} \equiv 0,
\end{align*}
\]

**Cfact S0**, default state, for the Cfacts T01, T10, T20.

[Axiom1] applied; of PFact 'Pfact-T00, '., A00, Root Initiator'

Actor 'A00, Root Initiator' with role 'Root Initiator' of PFact 'Pfact-T00, '
executing [Axiom1] selected option 'Yes' of 'requests 'Yes | No''.

[Cfact S1]
\[
\begin{align*}
&Rq_{T01} \equiv 1, \\
&Pm_{T01} \equiv 0, \\
&St_{T01} \equiv 0, \\
&Ac_{T01} \equiv 0, \\
&Rj_{T01} \equiv 0, \\
&Sp_{T01} \equiv 0,
\end{align*}
\]

**Cfact S1** a Request issued by the Initiator A00 for T00.

[Axiom5] applied; of PFact 'Pfact-T00, '., A01, '
Actor 'A01, ' with role 'Executor'; of PFact 'Pfact-T00, '
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

[Cfact S3]
\[
\begin{align*}
&Rq_{T01} \equiv 0, \\
&Pm_{T01} \equiv 1, \\
&Dc_{T01} \equiv 0, \\
&St_{T01} \equiv 0, \\
&Ac_{T01} \equiv 0, \\
&Rj_{T01} \equiv 0,
\end{align*}
\]

**Cfact S3** a Promise issued by the Executor A01 for T00.

[Axiom3] applied; Pfact-T10

[Cfact S1]
\[
\begin{align*}
&Rq_{T10} \equiv 1, \\
&Pm_{T10} \equiv 0, \\
&St_{T10} \equiv 0, \\
&Ac_{T10} \equiv 0, \\
&Rj_{T10} \equiv 0, \\
&Sp_{T10} \equiv 0,
\end{align*}
\]

**Cfact S1** a Request issued by the Initiator A01 for T10.

[Axiom5] applied; of PFact 'Pfact-T10, '., A10, '
Actor 'A10, ' with role 'Executor'; of PFact 'Pfact-T10, '
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.
Appendix I. Processor Simulation, Verification and Validation

[Cfact S3]
\[
\begin{align*}
  Rq &= 0 \\
  Pm &= 1 \\
  Dc, Qt &= 0 \\
  St, Ac, Rj, Sp &= 0
\end{align*}
\]

\textbf{Cfact S3} a Promise issued by the Executor A10 for T10.

[Axiom3] applied; Pfact-T20

[Cfact S1]
\[
\begin{align*}
  Rq &= 1 \\
  Pm, Dc, Qt &= 0 \\
  St, Ac, Rj, Sp &= 0
\end{align*}
\]

\textbf{Cfact S1} a Request issued by the Initiator A10 for T20.

[Axiom5] applied; of Pfact 'Pfact-T20, '., A20, '

Actor 'A20, ' with role 'Executor'; of Pfact 'Pfact-T20, '.
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

[Cfact S3]
\[
\begin{align*}
  Rq &= 0 \\
  Pm &= 1 \\
  Dc, Qt &= 0 \\
  St, Ac, Rj, Sp &= 0
\end{align*}
\]

\textbf{Cfact S3} a Promise issued by the Executor A20 for T20.

[Axiom21] applied; of Pfact 'Pfact-T20, '., A20, '

Actor 'A20, ' with role 'Executor'; of Pfact 'Pfact-T20, '.
executing [Axiom21] selected option 'State' of 'Executed State'.

[Cfact S6]
\[
\begin{align*}
  Rq &= 0 \\
  Pm &= 1 \\
  Dc, Qt &= 0 \\
  St &= 1 \\
  Ac, Rj, Sp &= 0
\end{align*}
\]

\textbf{Cfact S6}, a State has been issued by Executor A20 for T20.

[Axiom24] applied; Pfact-T20

Actor 'A10, ' with role 'Initiator'; of Pfact 'Pfact-T20, '.
executing [Axiom24-25] selected option 'Accept' of 'Accept | Reject'.

[Axiom2]
\[
\begin{align*}
  Rq &= 0 \\
  Pm &= 1 \\
  Dc, Qt &= 0 \\
  St, Ac, Rj, Sp &= 0
\end{align*}
\]

\textbf{Cfact S2} a Request issued by the Initiator A20 for T20.
Appendix I. \( \Psi \) Processor Simulation, Verification and Validation

\[\text{Cfact S11}\]
\[
\begin{align*}
Rq &= 0 \\
Pm &= 1 \\
Dc, Qt &= 0 \\
St &= 0 \\
Ac &= 1 \\
Rj, Sp &= 0
\end{align*}
\]

Cfact S11, Initiator A10 issued an Accept for T20.

[Axiom21] applied; of PFact 'Pfact-T10, '., A10, '
Actor 'A10, ' with role 'Executor'; of PFact 'Pfact-T10, '. executing [Axiom21] selected option 'State' of 'Executed State'.

\[\text{Cfact S6}\]
\[
\begin{align*}
Rq &= 0 \\
Pm &= 1 \\
Dc, Qt &= 0 \\
St &= 1 \\
Ac, Rj, Sp &= 0
\end{align*}
\]

Cfact S6, a State issued by the Executor A10 for T10.

[Axiom24] applied; Pfact-T10
Actor 'A01, ' with role 'Initiator'; of PFact 'Pfact-T10, '. executing [Axiom24-25] selected option 'Accept' of 'Accept | Reject'.

\[\text{Cfact S11}\]
\[
\begin{align*}
Rq &= 0 \\
Pm &= 1 \\
Dc, Qt &= 0 \\
St &= 0 \\
Ac &= 1 \\
Rj, Sp &= 0
\end{align*}
\]

Cfact S11, Initiator A01 issued an Accept for T10.

[Axiom21] applied; of PFact 'Pfact-T00, '., A01, '
Actor 'A01, ' with role 'Executor'; of PFact 'Pfact-T00, '. executing [Axiom21] selected option 'State' of 'Executed State'.

\[\text{Cfact S6}\]
\[
\begin{align*}
Rq &= 0 \\
Pm &= 1 \\
Dc, Qt &= 0 \\
St &= 1 \\
Ac, Rj, Sp &= 0
\end{align*}
\]

Cfact S6, a State has been issued by the Executor A01 for T00.

[Axiom24] applied; Pfact-T00
Appendix I. $\Psi$ Processor Simulation, Verification and Validation

Actor 'A00, Root Initiator' with role 'ROOT Initiator'; of PFact 'Pfact-T00',
executing [Axiom24-25] selected option 'Accept' of 'Accept | Reject'.

[Cfact S11]
\[
\begin{align*}
\text{Rq, Pm, Dc, Qt} & \equiv \phi \\
\text{St} & \equiv \phi \\
\text{Ac} & \equiv 1 \\
\text{Rj, Sp} & \equiv \phi
\end{align*}
\]

Cfact S11, Initiator A00 issued an Accept for T00.

The execution of this model in this way resulted in the performance and acceptance of T00.

Appendix I.4 Simulation of a branched Aggregated Model

In this section, a sufficiently complex branched aggregated $\Psi$ model is investigated as example and some of the phenomena described in Section 6 are shown. The part of interest is the role of Actor A10, with 2 child transactions T20 and T21. This fork shows rollback phenomena. It is clear from this model that the production (meaning Accepted == true, Cfact S11) of Root Pfact of Transaction T00 requires the prior production of the Pfact of Transaction T10. Similarly, the Pfact of Transaction T10 requires the prior production of both the Pfacts of Transactions T20 and T21. The productions of T20 and T30 require the prior productions of Transactions T30 and T31 respectively and the production T31 requires prior production of Transaction T41.
Figure 8.3. An aggregated \( \Psi \) model with a branched tree structure.

The model represented as NLDMOL Aspect model:

```
DEMO model 'DEMOApplication' 'Test version DEMO Tree'.
Declaration of Actors.
Root Actor-Initiator 'A00' 'Root Initiator';
Actor 'A01';
Actor 'A10';
Actor 'A20';
Actor 'A21';
Actor 'A30';
Actor 'A31';
Actor 'A41';

Declaration of Transactions.
Transaction 'Pfact-T00';
Transaction 'Pfact-T10';
Transaction 'Pfact-T20';
Transaction 'Pfact-T30';
Transaction 'Pfact-T31';
Transaction 'Pfact-T41';

Declaration of Transaction aggregation.
```
Actor-Initiator 'A00' 'Root Initiator'
  requires from:
  Actor-Executor 'A01' performing 'Pfact-T00'
  requires from:
  Actor-Executor 'A10' performing 'Pfact-T10'
  requires from:
  Actor-Executor 'A20' performing 'Pfact-T20'
  requires from:
  Actor-Executor 'A30' performing 'Pfact-T30'
  end of require;
  Actor-Executor 'A21' performing 'Pfact-T21'
  requires from:
  Actor-Executor 'A31' performing 'Pfact-T31'
  requires from:
  Actor-Executor 'A41' performing 'Pfact-T41'
  end of require;
  end of require;
  end of require;
  end of require;
  end of require.

Declaration of Transaction states.
Actor-Initiator 'A00' 'Root Initiator' for Transaction 'Pfact-T00' has Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A01' .
Query Actor-Initiator 'Request Yes|No'.

Actor-Initiator 'A01' for Transaction 'Pfact-T10' has Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A10' .

Actor-Initiator 'A10' for Transaction 'Pfact-T20' has Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A20' .

Actor-Initiator 'A20' for Transaction 'Pfact-T30' has Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A30' .

Actor-Initiator 'A10' for Transaction 'Pfact-T21' has Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
Stated = false; Rejected = false; Stopped = false; Accepted = false; with Actor-Executor 'A21'.

Actor-Initiator 'A21' for Transaction 'Pfact-T31' has Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A31'.

Actor-Initiator 'A31' for Transaction 'Pfact-T41' has Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A41'.

The rendered $\Psi$ model of fig. 8.3 in DMOL representation (after automated white space formatting):

```xml
<?xml version="1.0" encoding="windows-1252" ?>
<DMApp>DEMOApplication
<PfactChilds>
  <Pfact>Pfact-T00
    <PfactPublic prmPfactIdentifier="Pfact-T00" prmPfactLabel=""
  </PfactPublic>
  </Pfact>
  <CFact>
    <CFactPrivate prmRequested="false" prmPromised="false"
      prmStated="false" prmAccepted="false" prmDeclined="false" prmQuited="false"
      prmRejected="false" prmStopped="false">
    </CFactPrivate>
    <CFactPublic prmCFactIdentifier="Cfact-T00" prmCFactLabel=""
    </CFactPublic>
  </CFact>
</PfactChilds>
<ActExec>
  <Act>
    <ActPublic prmActIdentifier="A01" prmActLabel=""
  </ActPublic>
</Act>
</ActExec>

<ActInit>
  <Act>
    <ActPublic prmActIdentifier="A00" prmActLabel="Root Initiator">
  </ActPublic>
</Act>
</Act>
```
Appendix I. $\Psi$ Processor Simulation, Verification and Validation

</ActInit>

<PfactChilds>
  <Pfact>Pfact-T10
  <PfactPublic prmPfactIdentifier="Pfact-T10" prmPfactLabel=""
  </PfactPublic>
  </Pfact>
  </PfactChilds>

<CFact>
  <CFactPrivate prmRequested="false" prmPromised="false"
  prmStated="false" prmAccepted="false" prmDeclined="false"
  prmRejected="false" prmStopped="false">
  </CFactPrivate>
  <CFactPublic prmCFactIdentifier="Cfact-T10" prmCFactLabel=""
  </CFactPublic>
</CFact>

<ActExec>
  <Act>
    <ActPublic prmActIdentifier="A10" prmActLabel=""
    </ActPublic>
  </Act>
</ActExec>

<PfactChilds>
  <Pfact>Pfact-T20
  <PfactPublic prmPfactIdentifier="Pfact-T20" prmPfactLabel=""
  </PfactPublic>
  </Pfact>
  </PfactChilds>

<CFact>
  <CFactPrivate prmRequested="false" prmPromised="false"
  prmStated="false" prmAccepted="false" prmDeclined="false"
  prmRejected="false" prmStopped="false">
  </CFactPrivate>
  <CFactPublic prmCFactIdentifier="Cfact-T20" prmCFactLabel=""
  </CFactPublic>
</CFact>

<ActExec>
  <Act>
    <ActPublic prmActIdentifier="A20" prmActLabel=""
    </ActPublic>
  </Act>
</ActExec>

<PfactChilds>
<Pfact>Pfact-T30
  <PfactPublicprmPfactIdentifier="Pfact-T30" prmPfactLabel=""/>
</PfactPublic>

<CFact>
  <CFactPrivate prmRequested="false" prmPromised="false" prmStated="false" prmAccepted="false" prmDeclined="false" prmQuited="false" prmRejected="false" prmStopped="false"/>
  <CFactPublic prmCFactIdentifier="CFact-T30" prmCFactLabel=""/>
</CFact>

<ActExec>
  <Act>
    <ActPublic prmActIdentifier="A30" prmActLabel=""/>
  </Act>
  <ActExec>
  <PfactChilds>
  </Pfact>

<Pfact>Pfact-T21
  <PfactPublic prmPfactIdentifier="Pfact-T21" prmPfactLabel=""/>
</PfactPublic>

<CFact>
  <CFactPrivate prmRequested="false" prmPromised="false" prmStated="false" prmAccepted="false" prmDeclined="false" prmQuited="false" prmRejected="false" prmStopped="false"/>
  <CFactPublic prmCFactIdentifier="Cfact-T21" prmCFactLabel=""/>
</CFact>

<ActExec>
  <Act>
    <ActPublic prmActIdentifier="A21" prmActLabel=""/>
  </Act>
  <ActExec>
  <PfactChilds>
  </Pfact>
Simulation VII
The simulation is about the DEMO model of fig. 8.4.1 There is no need to show the
standard transaction completion (Request-Promise-State-Accept). Instead in this
simulation some of the phenomena of section 3.3. are shown. The execution of this example
is as follows:
The cycle Request – Promise continues until A30 and A41 issue a Promise and are both
about to issue a State.
Transaction T41 is Stated by A41 and Accepted by A31.
Actor A31 issues a State for Transaction A31 and waits.
Actor A30 issues a State for transaction T30, followed by a Reject by Actor A20 and a Stop
from Actor A30. Transaction T30 fails and the entire model should 'recover' from that
condition.

Note: In an aggregated Model there are typically multiple state transitions possible at any
time. In some cases there are multiple options for the same Actor, though in different roles.

The following logging results are shown.
Comments are inserted as /* Comments */.
Actor options are traced as follows:

```
ActorQuery Actor = A30, ' ; with role 'Executor'; Verb = 'Promise | Decline'; of PFact 'Pfact-T30, '.
```

In this example Actor A30, with role Executor has the option to Promise or Decline for
PFact-T30. Actor A30 may or may not choose one of these options. Multiple options for
multiple Actors are common in such a DEMO model. In the previous simulations the Actor
option logging was switched off.

Execution tracing starting from the default state, all Cfact state are Cfact S0, results then in:

```
ActorQuery Actor = A00, Root Initiator' ; with role 'Root Initiator'
Verb = 'requests 'Yes | No''; of PFact 'Pfact-T00, '.
/* Actor A00, as Initiator has the option to issue a Request ( or not )
for Pfact-T00. */
```
Appendix I. \( \Psi \) Processor Simulation, Verification and Validation

[Axiom1] applied; of PFact 'Pfact-T00, ., A00, Root Initiator'

Actor 'A00, Root Initiator' with role 'Root Initiator' of PFact 'Pfact-T00, .',
executing [Axiom1] selected option 'Yes' of 'requests 'Yes | No'.'

[Cfact S1]

\[
\begin{align*}
R_q &= 1 \\
P_m, D_c, Q_t &= 0 \\
S_t, A_c, R_j, S_p &= 0
\end{align*}
\]

Cfact S1 a Request issued by the Initiator A00 for T00.

ActorQuery Actor = A01, ' ; with role 'Executor'; Verb = 'Promise | Decline'; of PFact 'Pfact-T00, .'

[Axiom5] applied; of PFact 'Pfact-T00, ., A01, '

Actor 'A01, ' with role 'Executor'; of PFact 'Pfact-T00, .'.
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

[Cfact S3]

\[
\begin{align*}
R_q &= 0 \\
P_m &= 1 \\
D_c, Q_t &= 0 \\
S_t, A_c, R_j, S_p &= 0
\end{align*}
\]

Cfact S3 a Promise issued by the Executor A01 for T00.

[Axiom3] applied; Pfact-T01

[Cfact S1]

\[
\begin{align*}
R_q &= 1 \\
P_m, D_c, Q_t &= 0 \\
S_t, A_c, R_j, S_p &= 0
\end{align*}
\]

Cfact S1 a Request issued by the Initiator A01 for T10.

ActorQuery Actor = A10, ' ; with role 'Executor'; Verb = 'Promise | Decline'; of PFact 'Pfact-T10, .'.


Actor 'A10, ' with role 'Executor'; of PFact 'Pfact-T10, .'.
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.
Appendix I. \( \Psi \) Processor Simulation, Verification and Validation

[Cfact S3]
\[
\begin{align*}
R_q &= 0 \\
P_m &= 1 \\
D_c, Q_t &= 0 \\
S_t, A_c, R_j, S_p &= 0
\end{align*}
\]

Cfact S3, a Promise has been issued by the Executor A10 for T10.

[Axiom3] applied; Pfact-T20

[Cfact S1]
\[
\begin{align*}
R_q &= 1 \\
P_m, D_c, Q_t &= 0 \\
S_t, A_c, R_j, S_p &= 0
\end{align*}
\]

Cfact S1, a Request has been issued by the Initiator A10 for T20.

[Axiom3] applied; Pfact-T21

[Cfact S1]
\[
\begin{align*}
R_q &= 1 \\
P_m, D_c, Q_t &= 0 \\
S_t, A_c, R_j, S_p &= 0
\end{align*}
\]

Cfact S1, a Request has been issued by the Initiator A10 for T21.

ActorQuery Actor = A20, '; with role 'Executor'; Verb = 'Promise | Decline'; of PFact 'Pfact-T20, '.

ActorQuery Actor = A21, '; with role 'Executor'; Verb = 'Promise | Decline'; of PFact 'Pfact-T21, '.

/* The DEMO Processor renders 2 queries to Actors A20 and A21, both with the option to Promise or Decline. Only one query can be executed at a time. */

[Axiom5] applied; of PFact 'Pfact-T20, '., A20, '

Actor 'A20, ' with role 'Executor'; of PFact 'Pfact-T20, '. executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.
Appendix I. Ψ Processor Simulation, Verification and Validation

[Cfact S3]

\[
\begin{align*}
& Rq = 0 \\
& Pm = 1 \\
& Dc,Qt = 0 \\
& St,Ac,Rj,Sp = 0
\end{align*}
\]

Cfact S3, a Promise has been issued by the Executor A20 for T20.

[Axiom3] applied; Pfact-T30

[Cfact S1]

\[
\begin{align*}
& Rq = 1 \\
& Pm, Dc, Qt = 0 \\
& St, Ac, Rj, Sp = 0
\end{align*}
\]

Cfact S1, a Request has been issued by the Initiator A20 for T30.

ActorQuery Actor = A30, ' ; with role 'Executor'; Verb = 'Promise | Decline'; of PFact 'Pfact-T30, '

ActorQuery Actor = A21, ' ; with role 'Executor'; Verb = 'Promise | Decline'; of PFact 'Pfact-T21, '


[Cfact S3]

\[
\begin{align*}
& Rq = 0 \\
& Pm = 1 \\
& Dc, Qt = 0 \\
& St, Ac, Rj, Sp = 0
\end{align*}
\]

Cfact S3, a Promise has been issued by the Executor A21 for T21.

ActorQuery Actor = A30, ' ; with role 'Executor'; Verb = 'Promise | Decline'; of PFact 'Pfact-T31, '

[Axiom3] applied; Pfact-T31
Cfact S1, a Request has been issued by the Initiator A21 for T31.

\begin{align*}
Rq & = 1 \\
Pt, DC, Qt & = 0 \\
St, Ac, Rj, Sp & = 0
\end{align*}

ActorQuery Actor = A31, ' ; with role 'Executor'; Verb = 'Promise | Decline'; of PFact 'Pfact-T31, '.

[Axiom5] applied; of PFact 'Pfact-T31, '., A31, '

Actor 'A31, ' with role 'Executor'; of PFact 'Pfact-T31, '
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

Cfact S3, a Promise has been issued by the Executor A31 for T31.

\begin{align*}
Rq & = 0 \\
Pt & = 1 \\
DC, Qt & = 0 \\
St, Ac, Rj, Sp & = 0
\end{align*}

ActorQuery Actor = A30, ' ; with role 'Executor'; Verb = 'Promise | Decline'; of PFact 'Pfact-T30, '

[Axiom3] applied; PFact-T41

Cfact S1, a Request has been issued by the Initiator A31 for T41.

\begin{align*}
Rq & = 1 \\
Pt, DC, Qt & = 0 \\
St, Ac, Rj, Sp & = 0
\end{align*}

ActorQuery Actor = A41, ' ; with role 'Executor'; Verb = 'Promise | Decline'; of PFact 'Pfact-T41, '

[Axiom5] applied; of PFact 'Pfact-T41, '., A41, '

Actor 'A41, ' with role 'Executor'; of PFact 'Pfact-T41, '.
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

Appendix I. Processor Simulation, Verification and Validation
Appendix I. Ψ Processor Simulation, Verification and Validation

[Cfact S3]
\[
\begin{align*}
R_q &= 0 \\
F_m &= 1 \\
D_c, Q_t &= 0 \\
S_t, A_c, R_j, S_p &= 0
\end{align*}
\]

Cfact S3, a Promise has been issued by the Executor A41 for T41.

/*
At this stage Actor A30 has the option to issue a Promise or a Decline for transaction T30.
Actor A41 has the option to issue a State for transaction T41.
*/

ActorQuery Actor = A30, ' ; with role 'Executor';  Verb = 'Promise | Decline'; of PFact 'Pfact-T30, '.

ActorQuery Actor = A41, ' ; with role 'Executor';  Verb = 'Executed State'; of PFact 'Pfact-T41, '.

[Axiom21] applied; of PFact 'Pfact-T41, '., A41, '
Actor 'A41, ' with role 'Executor'; of PFact 'Pfact-T41, '.
executing [Axiom21] selected option 'State' of 'Executed State'.

[Cfact S6]
\[
\begin{align*}
R_q &= 0 \\
F_m &= 1 \\
D_c, Q_t &= 0 \\
S_t &= 1 \\
A_c, R_j, S_p &= 0
\end{align*}
\]

Cfact S6, a State has been issued by the Executor A41 for Transaction T41.

ActorQuery Actor = A30, ' ; with role 'Executor';  Verb = 'Promise | Decline'; of PFact 'Pfact-T30, '.

ActorQuery Actor = A31, ' ; with role 'Initiator';  Verb = 'Accept | Reject'; of PFact 'Pfact-T41, '.

[Axiom24] applied; Pfact-T41
Actor 'A31, ' with role 'Initiator'; of PFact 'Pfact-T41, '.
executing [Axiom24-25] selected option 'Accept' of 'Accept | Reject'.
**Cfact S11**

\[
\begin{align*}
R_g & = 0 \\
P_m & = 1 \\
D_c & = 0 \\
Q_t & = 0
\end{align*}
\]

*Cfact S11*, Initiator A31 issued an Accept for transaction T41; transaction completed.

/*
Transaction T41 has completed.
The simulation continues by letting in the other branch transaction T30 go through a Promise - State - Reject - Stop sequence.
This represents the failed production of some component.
*/

ActorQuery Actor = A30, ' ; with role 'Executor'; Verb = 'Promise' | Decline'; of PFact 'Pfact-T30, '.

ActorQuery Actor = A31, ' ; with role 'Executor'; Verb = 'Executed State'; of PFact 'Pfact-T31, '.

[Axiom5] applied; of PFact 'Pfact-T30, '.', A30, '

Actor 'A30, ' with role 'Executor'; of PFact 'Pfact-T30, '. executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise' | Decline'.

**Cfact S3**

\[
\begin{align*}
R_g & = 0 \\
P_m & = 1 \\
D_c & = 0 \\
Q_t & = 0
\end{align*}
\]

*Cfact S3*, a Promise has been issued by the Executor A30 for T30.

ActorQuery Actor = A30, ' ; with role 'Executor'; Verb = 'Executed State'; of PFact 'Pfact-T30, '.

ActorQuery Actor = A31, ' ; with role 'Executor'; Verb = 'Executed State'; of PFact 'Pfact-T31, '.

[Axiom21] applied; of PFact 'Pfact-T30, '.', A30, '

Actor 'A30, ' with role 'Executor'; of PFact 'Pfact-T30, '. executing [Axiom21] selected option 'State' of 'Executed State'.

ActorQuery Actor = A31, ' ; with role 'Executor'; Verb = 'Executed State'; of PFact 'Pfact-T31, '.

Actor 'A31, ' with role 'Executor'; of PFact 'Pfact-T31, '. executing [Axiom21] selected option 'State' of 'Executed State'.
Appendix I. Ψ Processor Simulation, Verification and Validation

[Cfact S6]

\[
\begin{align*}
R_q &= 0 \\
P_m &= 1 \\
D_c \cdot Q_t &= 0 \\
S_t &= 1 \\
A_c, R_j, S_p &= 0 \\
\end{align*}
\]

Cfact S6, a State has been issued by the Executor A30 for Transaction T30.

ActorQuery Actor = A20, ' with role 'Initiator'; Verb = 'Accept | Reject'; of PFact 'Pfact-T30, '.

ActorQuery Actor = A31, ' with role 'Executor'; Verb = 'Executed State'; of PFact 'Pfact-T31, '.

[Axiom25] applied; Pfact-T30

Actor 'A20, ' with role 'Initiator'; of PFact 'Pfact-T30, '. executing [Axiom24-25] selected option 'Reject' of 'Accept | Reject'.

[Cfact S7]

\[
\begin{align*}
R_q &= 0 \\
P_m &= 1 \\
D_c \cdot Q_t &= 0 \\
S_t &= 1 \\
A_c &= 0 \\
R_j &= 1 \\
S_p &= 0 \\
\end{align*}
\]

Cfact S7, Initiator A20 issued a Reject for transaction T30.

ActorQuery Actor = A30, ' with role 'Executor'; Verb = 'State | Stop'; of PFact 'Pfact-T30, '.

ActorQuery Actor = A31, ' with role 'Executor'; Verb = 'Executed State'; of PFact 'Pfact-T31, '.


[Cfact S9]

\[
\begin{align*}
R_q P_m D_c Q_t &= 0 \\
S_t A_c R_j &= 0 \\
S_p &= 1 \\
\end{align*}
\]

Cfact S9, Executor A30 issued a Stop for transaction T30.
Appendix I. \( \Psi \) Processor Simulation, Verification and Validation

ActorQuery Actor = A20, ' ; with role 'Initiator'; Verb = 'Stop Acknowledge'; of PFact 'Pfact-T30, '.

ActorQuery Actor = A31, ' ; with role 'Executor'; Verb = 'Executed State'; of PFact 'Pfact-T31, '.

[Axiom29] applied; Pfact-T20

[Axiom28] applied; Pfact-T30

Actor 'A20, ' with role 'Initiator'; of PFact 'Pfact-T30, '.
executing [Axiom28-29-30-31-32-33] selected option 'Acknowledge' of 'Stop Acknowledge'.

\[
\begin{array}{l}
\text{Cfact S2, Actor A20 issued a Decline for transaction T20.}
\end{array}
\]

This is not a violation of the transaction axiom. The executor issued first a Promise but then replaces this Promise by a Decline, which is allowed since the state of the transaction did not change and the Initiator did not react on this Promise.

\[
\begin{array}{l}
\text{Cfact S0, Actor A20 restored the default state for transaction T30.}
\end{array}
\]

From here the upward propagation and roll-back of a failed production through the Quit – Quit-acknowledge – Decline – Quit mechanism is shown.

\[
\begin{array}{l}
\text{ActorQuery Actor = A10, ' ; with role 'Initiator'; Verb = 'Request| Quit'; of PFact 'Pfact-T20, '.
\end{array}
\]

ActorQuery Actor = A31, ' ; with role 'Executor'; Verb = 'Executed State'; of PFact 'Pfact-T31, '.

[Axiom8] applied; Actor-Executor A10, '

[Axiom9] applied; Pfact-T20
Appendix I. \( \Psi \) Processor Simulation, Verification and Validation

[Axiom11] applied; Pfact-T21

Actor 'A10, ' with role 'Initiator'; of PFact 'Pfact-T20, '.
executing [Axiom7-8-9-10-11-12-13] selected option 'Quit' of 'Request| Quit'.

/*
The choice of Actor A10 for the option Quit of "Request-or-Quit" invokes
execution of 3 axioms, [Axiom8], [Axiom9] and [Axiom11]; hence these 3
state transitions issued by Actor A10:
*/

[Cfact S2]

\( \begin{align*}
R_q & = 0 \\
P_m & = 0 \\
D_c & = 0 \\
O_t & = 0 \\
S_t, A_c, R_j, S_p & = 0
\end{align*} \)  \hspace{1cm} Cfact S2, Decline issued by Executor A10 for T10; [Axiom8].

[Cfact S5]

\( \begin{align*}
R_q & = 0 \\
P_m & = 0 \\
D_c, O_t & = 0 \\
S_t, A_c, R_j, S_p & = 0
\end{align*} \)  \hspace{1cm} Cfact S5, Initiator A10 issued a Quit for transaction T20;

[Cfact S4]

\( \begin{align*}
R_q & = 0 \\
P_m & = 0 \\
D_c & = 0 \\
O_t & = 0 \\
S_t, A_c, R_j, S_p & = 0
\end{align*} \)  \hspace{1cm} Cfact S4, Initiator A10 issued a Quit for transaction T21;

[Axiom11].

/*
At this point there are 4 simultaneous Queries issued by the Processor:
*/

ActorQuery Actor = A01, ' ; with role 'Initiator'; Verb = 'Request| Quit'; of PFact 'Pfact-T10, '.

ActorQuery Actor = A20, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of PFact 'Pfact-T20, '.

ActorQuery Actor = A21, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of PFact 'Pfact-T21, '.
Appendix I. ψ Processor Simulation, Verification and Validation

ActorQuery Actor = A31, ' ; with role 'Executor'; Verb = 'Executed State'; of PFact 'Pfact-T31, '.

[Axiom14] applied; of PFact 'Pfact-T20, '., A20, '
Actor 'A20, ' with role 'Executor'; of PFact 'Pfact-T20, '.
executing Axiom14-15-16-17-18-19] selected option 'Acknowledge' of 'Quit - Acknowledge'.

[Cfact S0]
\[
\begin{align*}
& Rq, Pm, Dc, Qt \equiv 0 \\
& St, Ac, Rj, Sp \equiv 0
\end{align*}
\]

Cfact S0, Actor A20 restored the default state for transaction T20; [Axiom14].

ActorQuery Actor = A01, ' ; with role 'Initiator'; Verb = 'Request| Quit'; of PFact 'Pfact-T10, '.

ActorQuery Actor = A21, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of PFact 'Pfact-T21, '.

ActorQuery Actor = A31, ' ; with role 'Executor'; Verb = 'Executed State'; of PFact 'Pfact-T31, '.

[Axiom8] applied; Actor-Executor A01, '
[Axiom9] applied; Pfact-T10
Actor 'A01, ' with role 'Initiator'; of PFact 'Pfact-T10, '.
executing [Axiom7-8-9-10-11-12-13] selected option 'Quit' of 'Request| Quit'.

[Cfact S2]
\[
\begin{align*}
& Rq, Pm \equiv 0 \\
& Dc \equiv 1 \\
& Qt \equiv 0 \\
& St, Ac, Rj, Sp \equiv 0
\end{align*}
\]

Cfact S2, a Decline has been issued by Executor A01 for T00; [Axiom8].

[Cfact S5]
\[
\begin{align*}
& Rq, Pm \equiv 0 \\
& Dc, Qt \equiv 1 \\
& St, Ac, Rj, Sp \equiv 0
\end{align*}
\]

Cfact S5, Initiator A01 issued a Quit for transaction T10; [Axiom9].

/*
At this point the upward roll-back Decline - Quit - Quit-Acknowledge cycle is visible.
ActorQuery Actor = A00; Root Initiator; with role 'Initiator'; Verb = 'Request| Quit'; of PFact 'Pfact-T00', '

ActorQuery Actor = A10, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of PFact 'Pfact-T10, '

ActorQuery Actor = A21, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of PFact 'Pfact-T21, '

ActorQuery Actor = A31, ' ; with role 'Executor'; Verb = 'Executed State'; of PFact 'Pfact-T31, '

[Axiom9] applied; Pfact-T00

Actor 'A00, Root Initiator' with role 'Initiator' of PFact 'Pfact-T00, '
executing [Axiom7-8-9-10-11-12-13-a] selected option 'Quit' of 'Request| Quit'.

[Cfact S5]
\[
\begin{align*}
Rq_{Pm} & \equiv 0 \\
Dc\_Qt & \equiv 1 \\
St\_Ac\_Rj\_Sp & \equiv 0
\end{align*}
\]

[Cfact S5], Initiator A00 issued a Quit for transaction T00;

[Axiom9].

ActorQuery Actor = A01, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of PFact 'Pfact-T00, '

ActorQuery Actor = A10, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of PFact 'Pfact-T10, '

ActorQuery Actor = A21, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of PFact 'Pfact-T21, '

ActorQuery Actor = A31, ' ; with role 'Executor'; Verb = 'Executed State'; of PFact 'Pfact-T31, '

[Axiom14] applied; of PFact 'Pfact-T00, ., A01, '

Actor 'A01, ' with role 'Executor'; of PFact 'Pfact-T00, '
executing Axiom14-15-16-17-18-19] selected option 'Acknowledge' of 'Quit - Acknowledge'.

[Cfact S0]
\[
\begin{align*}
Rq_{Pm}, Dc\_Qt & \equiv 0 \\
St\_Ac\_Rj\_Sp & \equiv 0
\end{align*}
\]

[Cfact S0, Actor A01 restored the default state for transaction T00; [Axiom14].
/* 
The Root transaction has returned to its default state due to the failure to deliver Pfact-T30. 
At this point the downward propagation of a Quit – Quit-Acknowledge – Quit for transactions T21, T31 and T41 continues, which should result in a return to default state S0 for all transactions except T41.
*/

ActorQuery Actor = A00, Root Initiator'; with role 'Root Initiator'
Verb = 'requests 'Yes | No''; of PFact 'Pfact-T00, '.

ActorQuery Actor = A10, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of PFact 'Pfact-T10, '.

ActorQuery Actor = A21, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of PFact 'Pfact-T21, '.

ActorQuery Actor = A31, ' ; with role 'Executor'; Verb = 'Executed State'; of PFact 'Pfact-T31, '.

[Axiom14] applied; of PFact 'Pfact-T10, ', A10, '
Actor 'A10, ' with role 'Executor'; of PFact 'Pfact-T10, '. executing Axiom14-15-16-17-18-19 selected option 'Acknowledge' of 'Quit - Acknowledge'.

[Cfact S0]
\[
\begin{cases}
Rq, Pm, Dc, Ot \rightarrow \circ \\
St, Ac, Rj, Sp \rightarrow \circ
\end{cases}
\]

Cfact S0, Actor A10 restored the default state for transaction T10; [Axiom14].

ActorQuery Actor = A00, Root Initiator'; with role 'Root Initiator'
Verb = 'requests 'Yes | No''; of PFact 'Pfact-T00, '.

/*/ 
Actor A00 has the option to restart the whole transaction tree since transaction T00 is in the default state again. */

ActorQuery Actor = A21, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of PFact 'Pfact-T21, '.

ActorQuery Actor = A31, ' ; with role 'Executor'; Verb = 'Executed State'; of PFact 'Pfact-T31, '.

[Axiom20] applied; of PFact 'Pfact-T21, ', A21, '
[Axiom36] applied; Pfact-T31
Actor 'A21, ' with role 'Executor'; of PFact 'Pfact-T21, '.
executing [Axiom20-34-35-36-37-38] selected option 'Acknowledge' of 'Quit - Acknowledge'.

[Cfact S0]
\[
\begin{align*}
R_q & = 0 \\
\bar{P}m & = 1 \\
D_c & = 0 \\
Q_t & = 1 \\
S_t, A_c, R_j, S_p & = 0
\end{align*}
\]

Cfact S0, Actor A21 restored the default state for transaction T21; [Axiom20].

[Cfact S4]
\[
\begin{align*}
R_q & = 0 \\
\bar{P}m & = 1 \\
D_c & = 0 \\
Q_t & = 1 \\
S_t, A_c, R_j, S_p & = 0
\end{align*}
\]

Cfact S4, Initiator A21 issued a Quit for transaction T31. [Axiom36].

ActorQuery Actor = A00, Root Initiator'; with role 'Root Initiator'
Verb = 'requests 'Yes | No''; of PFact 'Pfact-T00, '.

ActorQuery Actor = A31, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of PFact 'Pfact-T31, '.

[Axiom20] applied; of PFact 'Pfact-T31, '., A31, '

Actor 'A31, ' with role 'Executor'; of PFact 'Pfact-T31, '.
executing [Axiom20-34-35-36-37-38] selected option 'Acknowledge' of 'Quit - Acknowledge'.

[Cfact S0]
\[
\begin{align*}
R_q & = 0 \\
\bar{P}m & = 1 \\
D_c & = 0 \\
Q_t & = 1 \\
S_t, A_c, R_j, S_p & = 0
\end{align*}
\]

Cfact S0, Executor A31 restored the default state for transaction T31; [Axiom20].

ActorQuery Actor = A00, Root Initiator'; with role 'Root Initiator'
Verb = 'requests 'Yes | No''; of PFact 'Pfact-T00, '.

/*
The transaction T41 is still in the state Cfact S11, Accept == true, while all other transactions have returned to the default state. The simulation may continue and be repeated from this point.*/
Appendix I.5 Verification of Cancellation Patterns

A sufficient complex aggregated $\Psi$ model is investigated to simulate the effect of a Cancel operation and rollback as described in Section 6. The production (meaning Accepted == true, Cfact state S11) of Root Pfact of Transaction T00 requires the prior production of the Pfact of Transaction T10. Similarly the Pfact of Transaction T10 requires the prior production of both the Pfacts of Transactions T20 and T21. The productions of T20 and T30 require the prior productions of Transactions T30 and T31 respectively and the production T31 requires prior production of Transaction T41.

![Diagram of 6 sequentially nested transactions](image)

**Figure 8.4.** Model composed of 6 sequentially nested transactions.

The simulation carried out involves the repeated Request-Promise cycle until Actor A50 issues a promise, followed by a State for Pfact P50, and an Accept from Actor A40. Transaction T50 has been completed.
The parent transactions are still in the Promised state, Cfact state S3.
At this point Actor A20 decides for some unknown reason for a Cancel operation.
The Cancel operation induces the upward Rollback cycle through transaction T20, until transaction T00 reaches the default state. The downward Rollback cycle is induced via transaction T30 until Actor A40. Because the transaction T50 has been completed – Accepted – no state change is possible here, but all other transaction states return finally to the default state S0.

In the following trace information, totally rendered by the processor without any manual editing, at some places comments /* … */ have been inserted.

DEM0 model 'DEM0Application' 'Test version DEMO Tree'.
Declaration of Actors.
Root Actor-Initiator 'Root Initiator' 'The customer';
Actor 'Company' 'The Enterprise';
Actor 'A10';
Actor 'A20';
Actor 'A30';
Actor 'A40';
Actor 'A50';

Declaration of Transactions.
Transaction 'RootPfact' 'Complete Performance';
Transaction 'P10';
Transaction 'P20';
Transaction 'P30';
Transaction 'P40';
Transaction 'P50';

Declaration of Transaction aggregation.
Actor-Initiator 'Root Initiator' 'The customer'
  requires from:
    Actor-Executor 'Company' 'The Enterprise' performing 'RootPfact' 'Complete Performance'
    requires from:
      Actor-Executor 'A10' performing 'P10'
      requires from:
        Actor-Executor 'A20' performing 'P20'
        requires from:
          Actor-Executor 'A30' performing 'P30'
          requires from:
            Actor-Executor 'A40' performing 'P40'
            requires from:
              Actor-Executor 'A50' performing 'P50'
              end of require;
            end of require;
        end of require;
      end of require;
    end of require;
  end of require;
end of require;
end of require;
end of require;
end of require.

Declaration of Transaction states.
Actor-Initiator 'Root Initiator' 'The customer' for Transaction 'RootPfact' 'Complete Performance' has Cfact state of:
- Requested = false; Promised = false; Declined = false; Quited = false;
- Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'Company' 'The Enterprise'.
Query Actor-Initiator 'Root Initiator' 'The customer'
with role 'Root Initiator' Verb = 'requests Yes | No' of Pfact 'RootPfact, Complete Performance'.

Actor-Initiator 'Company' 'The Enterprise' for Transaction 'P10' has Cfact state of:
- Requested = false; Promised = false; Declined = false; Quited = false;
- Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A10'.

Actor-Initiator 'A10' for Transaction 'P20' has Cfact state of:
- Requested = false; Promised = false; Declined = false; Quited = false;
- Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A20'.

Actor-Initiator 'A20' for Transaction 'P30' has Cfact state of:
- Requested = false; Promised = false; Declined = false; Quited = false;
- Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A30'.

Actor-Initiator 'A30' for Transaction 'P40' has Cfact state of:
- Requested = false; Promised = false; Declined = false; Quited = false;
- Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A40'.

Actor-Initiator 'A40' for Transaction 'P50' has Cfact state of:
- Requested = false; Promised = false; Declined = false; Quited = false;
- Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A50'.

ActorQuery Actor = Root Initiator, The customer' ; with role 'Root Initiator' Verb = 'requests Yes | No'; of Pfact 'RootPfact, Complete Performance'.
[Axiom1] applied; of Pfact 'RootPfact, Complete Performance'., Root Initiator, The customer'

Actor 'Root Initiator, The customer' with role 'Root Initiator' of Pfact 'RootPfact, Complete Performance'.
executing [Axiom1] selected option 'Yes' of 'requests Yes | No'.

ActorQuery Actor = Company, The Enterprise' ; with role 'Executor'; Verb = 'Promise | Decline'; of Pfact 'RootPfact, Complete Performance'.
[Axiom5] applied; of Pfact 'RootPfact, Complete Performance'., Company, The Enterprise'

Actor 'Company, The Enterprise' with role 'Executor'; of Pfact 'RootPfact, Complete Performance'.
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

Appendix I. Ψ Processor Simulation, Verification and Validation 318
Appendix I. Ψ Processor Simulation, Verification and Validation

[Axiom3] applied; P10

ActorQuery Actor = A10, ' ; with role 'Executor'; Verb = 'Promise | Decline'; of Pfact 'P10, '.

[Axiom5] applied; of Pfact 'P10, ', A10, '

Actor 'A10, ' with role 'Executor'; of Pfact 'P10, '.
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

[Axiom3] applied; P20

ActorQuery Actor = A20, ' ; with role 'Executor'; Verb = 'Promise | Decline'; of Pfact 'P20, '.

[Axiom5] applied; of Pfact 'P20, ', A20, '

Actor 'A20, ' with role 'Executor'; of Pfact 'P20, '.
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

[Axiom3] applied; P30

ActorQuery Actor = A30, ' ; with role 'Executor'; Verb = 'Promise | Decline'; of Pfact 'P30, '.

[Axiom5] applied; of Pfact 'P30, ', A30, '

Actor 'A30, ' with role 'Executor'; of Pfact 'P30, '.
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

[Axiom3] applied; P40

ActorQuery Actor = A40, ' ; with role 'Executor'; Verb = 'Promise | Decline'; of Pfact 'P40, '.

[Axiom5] applied; of Pfact 'P40, ', A40, '

Actor 'A40, ' with role 'Executor'; of Pfact 'P40, '.
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

[Axiom3] applied; P50

ActorQuery Actor = A50, ' ; with role 'Executor'; Verb = 'Promise | Decline'; of Pfact 'P50, '.

[Axiom5] applied; of Pfact 'P50, ', A50, '

Actor 'A50, ' with role 'Executor'; of Pfact 'P50, '.
executing [Axiom5][Axiom6] selected option 'Promise' of 'Promise | Decline'.

ActorQuery Actor = A50, ' ; with role 'Executor'; Verb = 'Executed State'; of Pfact 'P50, '.

[Axiom21] applied; of Pfact 'P50, ', A50, '

Actor 'A50, ' with role 'Executor'; of Pfact 'P50, '.


executing \([\text{Axiom}21]\) selected option 'State' of 'Executed State'.

\[
\text{ActorQuery Actor = A40, ' ; with role 'Initiator'; Verb = 'Accept | Reject'; of Pfact 'P50, '}. \\
\text{[Axiom24] applied; P50}
\]

Actor 'A40, ' with role 'Initiator'; of Pfact 'P50, '.
executing \([\text{Axiom}24-25]\) selected option 'Accept' of 'Accept | Reject'.

\[
\text{ActorQuery Actor = A40, ' ; with role 'Executor'; Verb = 'Executed State'; of Pfact 'P40, '}. \\
\]

At this point the nested transaction execution resulted in the Promise, State and Accept for Pfact P50. This transaction has completed with an accepted performance.

The current state of the model is:

\[
\text{DEMO model 'DEMOApplication' 'Test version DEMO Tree'.}
\]

Declaration of Actors.
Root Actor-Initiator 'Root Initiator' 'The customer';
Actor 'Company' 'The Enterprise';
Actor 'A10';
Actor 'A20';
Actor 'A30';
Actor 'A40';
Actor 'A50';

Declaration of Transactions.
Transaction 'RootPfact' 'Complete Performance';
Transaction 'P10';
Transaction 'P20';
Transaction 'P30';
Transaction 'P40';
Transaction 'P50';

Declaration of Transaction aggregation.
Actor-Initiator 'Root Initiator' 'The customer' requires from:
Actor-Executor 'Company' 'The Enterprise' performing 'RootPfact' 'Complete Performance'
requires from:
Actor-Executor 'A10' performing 'P10'
requires from:
Actor-Executor 'A20' performing 'P20'
requires from:
Actor-Executor 'A30' performing 'P30'
requires from:
Actor-Executor 'A40' performing 'P40'
requires from:
Actor-Executor 'A50' performing 'P50'
end of require;
end of require;
Declaration of Transaction states.

Actor-Initiator 'Root Initiator' 'The customer' for Transaction 'RootPfact' 'Complete Performance' has Cfact state of:
Requested = false; Promised = true; Declined = false; Quited = false;
Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'Company' 'The Enterprise'.
Query Actor-Executor 'State - Acknowledge'.

Actor-Initiator 'Company' 'The Enterprise' for Transaction 'P10' has Cfact state of:
Requested = false; Promised = true; Declined = false; Quited = false;
Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A10'.
Query Actor-Executor 'State - Acknowledge'.

Actor-Initiator 'A10' for Transaction 'P20' has Cfact state of:
Requested = false; Promised = true; Declined = false; Quited = false;
Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A10'.
Query Actor-Executor 'State - Acknowledge'.

Actor-Initiator 'A20' for Transaction 'P30' has Cfact state of:
Requested = false; Promised = true; Declined = false; Quited = false;
Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A30'.
Query Actor-Executor 'State - Acknowledge'.

Actor-Initiator 'A30' for Transaction 'P40' has Cfact state of:
Requested = false; Promised = true; Declined = false; Quited = false;
Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A40'.
Query Actor-Executor 'State - Acknowledge'.

Actor-Initiator 'A40' for Transaction 'P50' has Cfact state of:
Requested = false; Promised = true; Declined = false; Quited = false;
Stated = false; Rejected = false; Stopped = false; Accepted = true;
with Actor-Executor 'A50'.

/*
At this point Actor A20 decides, for some unknown reason, to issue a Cancel,
which induced the upward Rollback and downward Rollback mechanisms.
*/

[Axiom40] CancelExecutor S3->S2 applied; P20

[Axiom49] CancelInitiator S3->S4 applied; P30

/*
Actor A20 issued a Cancel.
As Executor of Pfact P20 the Cancel resulted in a Decline, according to
[Axiom40].
*/
As Initiator of Pfact P30 the Cancel resulted in a Quit, according to [Axiom49].
The upward Rollback mechanism is executed first:

/*

ActorQuery Actor = A10, ' ; with role 'Initiator'; Verb = 'Request| Quit';
of Pfact 'P20, '.

ActorQuery Actor = A30, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P30, '.

ActorQuery Actor = A40, ' ; with role 'Executor'; Verb = 'Executed State';
of Pfact 'P40, '.

[Axiom8] applied; Actor-Executor A10, '

[Axiom9] applied; P20
Actor 'A10, ' with role 'Initiator'; of Pfact 'P20, '.
executing [Axiom7-8-9-10-11-12-13] selected option 'Quit' of 'Request| Quit'.

ActorQuery Actor = Company, The Enterprise' ; with role 'Initiator'; Verb = 'Request| Quit'; of Pfact 'P10, '.

ActorQuery Actor = A20, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P20, '.

ActorQuery Actor = A30, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P30, '.

ActorQuery Actor = A40, ' ; with role 'Executor'; Verb = 'Executed State';
of Pfact 'P40, '.

[Axiom14] applied; of Pfact 'P20, ', A20, '
Actor 'A20, ' with role 'Executor'; of Pfact 'P20, '.
executing Axiom14-15-16-17-18-19] selected option 'Acknowledge' of 'Quit - Acknowledge'.

ActorQuery Actor = Company, The Enterprise' ; with role 'Initiator'; Verb = 'Request| Quit'; of Pfact 'P10, '.

ActorQuery Actor = A30, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P30, '.

ActorQuery Actor = A40, ' ; with role 'Executor'; Verb = 'Executed State';
of Pfact 'P40, '.

[Axiom8] applied; Actor-Executor Company, The Enterprise'

[Axiom9] applied; P10
Actor 'Company, The Enterprise' with role 'Initiator'; of Pfact 'P10, '.
executing [Axiom7-8-9-10-11-12-13] selected option 'Quit' of 'Request|Quit'.

ActorQuery Actor = Root Initiator, The customer' with role 'Initiator' Verb = 'Request|Quit'; of Pfact 'RootPfact, Complete Performance'.

ActorQuery Actor = A10, ' with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P10, '.

ActorQuery Actor = A30, ' with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P30, '.

ActorQuery Actor = A40, ' with role 'Executor'; Verb = 'Executed State'; of Pfact 'P40, '.

[Axiom9] applied; RootPfact
Actor 'Root Initiator, The customer' with role 'Initiator' of Pfact 'RootPfact, Complete Performance'.
executing [Axiom7-8-9-10-11-12-13-a] selected option 'Quit' of 'Request|Quit'.

ActorQuery Actor = Company, The Enterprise' with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'RootPfact, Complete Performance'.

ActorQuery Actor = A10, ' with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P10, '.

ActorQuery Actor = A30, ' with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P30, '.

ActorQuery Actor = A40, ' with role 'Executor'; Verb = 'Executed State'; of Pfact 'P40, '.

[Axiom14] applied; of Pfact 'P10, ', A10, '
Actor 'A10, ' with role 'Executor'; of Pfact 'P10, '.
executing Axiom14-15-16-17-18-19] selected option 'Acknowledge' of 'Quit - Acknowledge'.

ActorQuery Actor = Company, The Enterprise' with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'RootPfact, Complete Performance'.

ActorQuery Actor = A30, ' with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P30, '.

ActorQuery Actor = A40, ' with role 'Executor'; Verb = 'Executed State'; of Pfact 'P40, '.

[Axiom14] applied; of Pfact 'RootPfact, Complete Performance'., Company, The Enterprise'
Actor 'Company, The Enterprise' with role 'Executor'; of Pfact 'RootPfact, Complete Performance'.
executing Axiom14-15-16-17-18-19] selected option 'Acknowledge' of 'Quit - Acknowledge'.

executing [Axiom7-8-9-10-11-12-13-a] selected option 'Quit' of 'Request|Quit'.

ActorQuery Actor = Root Initiator, The customer' with role 'Initiator' Verb = 'Request|Quit'; of Pfact 'RootPfact, Complete Performance'.

ActorQuery Actor = A10, ' with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P10, '.

ActorQuery Actor = A30, ' with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P30, '.

ActorQuery Actor = A40, ' with role 'Executor'; Verb = 'Executed State'; of Pfact 'P40, '

[Axiom9] applied; RootPfact
Actor 'Root Initiator, The customer' with role 'Initiator' of Pfact 'RootPfact, Complete Performance'.
executing [Axiom7-8-9-10-11-12-13-a] selected option 'Quit' of 'Request|Quit'.

ActorQuery Actor = Company, The Enterprise' with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'RootPfact, Complete Performance'.

ActorQuery Actor = A10, ' with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P10, '.

ActorQuery Actor = A30, ' with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P30, '.

ActorQuery Actor = A40, ' with role 'Executor'; Verb = 'Executed State'; of Pfact 'P40, '

[Axiom14] applied; of Pfact 'P10, ', A10, '
Actor 'A10, ' with role 'Executor'; of Pfact 'P10, '.
executing Axiom14-15-16-17-18-19] selected option 'Acknowledge' of 'Quit - Acknowledge'.

ActorQuery Actor = Company, The Enterprise' with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'RootPfact, Complete Performance'.

ActorQuery Actor = A30, ' with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P30, '.

ActorQuery Actor = A40, ' with role 'Executor'; Verb = 'Executed State'; of Pfact 'P40, '

[Axiom14] applied; of Pfact 'RootPfact, Complete Performance'., Company, The Enterprise'
Actor 'Company, The Enterprise' with role 'Executor'; of Pfact 'RootPfact, Complete Performance'.
executing Axiom14-15-16-17-18-19] selected option 'Acknowledge' of 'Quit - Acknowledge'.
Appendix I. \( \Psi \) Processor Simulation, Verification and Validation

ActorQuery Actor = Root Initiator, The customer'; with role 'Root Initiator' Verb = 'requests 'Yes | No'' of Pfact 'RootPfact, Complete Performance'.

ActorQuery Actor = A30, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P30, '.

ActorQuery Actor = A40, ' ; with role 'Executor'; Verb = 'Executed State'; of Pfact 'P40, '.

The complete upward Rollback has been executed while for the downward Rollback so far nothing happened.

The current state of the model is:

DEMO model 'DEMOApplication' 'Test version DEMO Tree'.

Declaration of Actors.
Root Actor-Initiator 'Root Initiator' 'The customer'; Actor 'Company' 'The Enterprise'; Actor 'A10'; Actor 'A20'; Actor 'A30'; Actor 'A40'; Actor 'A50';

Declaration of Transactions.
Transaction 'RootPfact' 'Complete Performance'; Transaction 'P10'; Transaction 'P20'; Transaction 'P30'; Transaction 'P40'; Transaction 'P50';

Declaration of Transaction aggregation.
Actor-Initiator 'Root Initiator' 'The customer'
  requires from:
  Actor-Executor 'Company' 'The Enterprise' performing 'RootPfact'
  'Complete Performance'
  requires from:
  Actor-Executor 'A10' performing 'P10'
  requires from:
  Actor-Executor 'A20' performing 'P20'
  requires from:
  Actor-Executor 'A30' performing 'P30'
  requires from:
  Actor-Executor 'A40' performing 'P40'
  requires from:
  Actor-Executor 'A50' performing 'P50'
  end of require;
  end of require;
  end of require;
  end of require;
  end of require;
Declaration of Transaction states.

Actor-Initiator 'Root Initiator' 'The customer' for Transaction 'RootPfact'
'Complete Performance' has Cfact state of :
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'Company' 'The Enterprise'.

Query Actor-Initiator 'Request Yes|No'.

Actor-Initiator 'Company' 'The Enterprise' for Transaction 'P10' has Cfact
state of :
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A10' .

Actor-Initiator 'A10' for Transaction 'P20' has Cfact state of :
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A20' .

Actor-Initiator 'A20' for Transaction 'P30' has Cfact state of :
  Requested = false; Promised = true; Declined = false; Quited = true;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A30' .

Query Actor-Executor 'Quit - Acknowledge'.

Actor-Initiator 'A30' for Transaction 'P40' has Cfact state of :
  Requested = false; Promised = true; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A40' .

Query Actor-Executor 'State - Acknowledge'.

Actor-Initiator 'A40' for Transaction 'P50' has Cfact state of :
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = true;
with Actor-Executor 'A50' .

/*
From here on the downward Rollback mechanism is executed:
*/

ActorQuery Actor = Root Initiator, The customer'; with role 'Root
Initiator'; Verb = 'requests Yes | No'; of Pfact 'RootPfact, Complete
Performance'.

ActorQuery Actor = A30, ';' with role 'Executor'; Verb = 'Quit -
Acknowledge'; of Pfact 'P30', '.

ActorQuery Actor = A40, ';' with role 'Executor'; Verb = 'Executed State';
of Pfact 'P40', '.

[Axiom20] applied; of Pfact 'P30', '.', A30, '

[Axiom36] applied; P40
Actor 'A30', with role 'Executor'; of Pfact 'P30',
exacting \[\text{Axiom20-34-35-36-37-38}\] selected option 'Acknowledge' of 'Quit - Acknowledge'.

ActorQuery Actor = Root Initiator, The customer'; with role 'Root Initiator' Verb = 'requests 'Yes | No''; of Pfact 'RootPfact, Complete Performance'.

ActorQuery Actor = A40, ' ; with role 'Executor'; Verb = 'Quit - Acknowledge'; of Pfact 'P40, '.

[Axiom20] applied; of Pfact 'P40, '., A40, '

Actor 'A40, ' with role 'Executor'; of Pfact 'P40, '.
exacting \[\text{Axiom20-34-35-36-37-38}\] selected option 'Acknowledge' of 'Quit - Acknowledge'.

ActorQuery Actor = Root Initiator, The customer'; with role 'Root Initiator' Verb = 'requests 'Yes | No''; of Pfact 'RootPfact, Complete Performance'.

/*
The downward Rollback mechanism has been completed:
*/

DEMO model 'DEMOApplication' 'Test version DEMO Tree'.
Declaration of Actors.
Root Actor-Initiator 'Root Initiator' 'The customer';
Actor 'Company' 'The Enterprise';
Actor 'A10';
Actor 'A20';
Actor 'A30';
Actor 'A40';
Actor 'A50';

Declaration of Transactions.
Transaction 'RootPfact' 'Complete Performance';
Transaction 'P10';
Transaction 'P20';
Transaction 'P30';
Transaction 'P40';
Transaction 'P50';

Declaration of Transaction aggregation.
Actor-Initiator 'Root Initiator' 'The customer' requires from:
Actor-Executor 'Company' 'The Enterprise' performing 'RootPfact' 'Complete Performance'
requires from:
Actor-Executor 'A10' performing 'P10'
requires from:
Actor-Executor 'A20' performing 'P20'
requires from:
Actor-Executor 'A30' performing 'P30'
requires from:
Actor-Executor 'A40' performing 'P40'
Appendix I. Processor Simulation, Verification and Validation

requires from:
    Actor-Executor 'A50' performing 'P50'
end of require;
end of require;
end of require;
end of require;
end of require;
end of require.

Declaration of Transaction states.
Actor-Initiator 'Root Initiator' 'The customer' for Transaction 'RootPfact'
'Complete Performance' has Cfact state of:
    Requested = false; Promised = false; Declined = false; Quited = false;
    Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'Company' 'The Enterprise'.
Query Actor-Initiator 'Request Yes|No'.

Actor-Initiator 'Company' 'The Enterprise' for Transaction 'P10' has Cfact state of:
    Requested = false; Promised = false; Declined = false; Quited = false;
    Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A10'.

Actor-Initiator 'A10' for Transaction 'P20' has Cfact state of:
    Requested = false; Promised = false; Declined = false; Quited = false;
    Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A20'.

Actor-Initiator 'A20' for Transaction 'P30' has Cfact state of:
    Requested = false; Promised = false; Declined = false; Quited = false;
    Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A30'.

Actor-Initiator 'A30' for Transaction 'P40' has Cfact state of:
    Requested = false; Promised = false; Declined = false; Quited = false;
    Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A40'.

Actor-Initiator 'A40' for Transaction 'P50' has Cfact state of:
    Requested = false; Promised = false; Declined = false; Quited = false;
    Stated = false; Rejected = false; Stopped = false; Accepted = true;
with Actor-Executor 'A50'.

The upward and downward Rollback mechanisms have been executed and completed.
For Pfact the transaction has been completed, Accepted = true.
All other transactions are in the default state S0, and can be re-executed at any time.
Appendix I.6 Conclusions

“What, me worry?”

Alfred E. Neuman

According to Alfred E. Neuman, American philosopher, there is no need to worry.

The simulations and validations exhibit no anomalies, show compliance to the transaction axiom and show compliance to the phenomena observed in aggregated $\Psi$ models. There is no claim for completeness in verification so a claim for formal correctness cannot be made. There is a reasonable degree of confidence of correctness. There is also ground for the confidence that, if any errors are found later, these errors can be corrected, resulting in a formal correct set of state transition functions.
Appendix II. \( \Psi \) Processor Implementation

This appendix is a discussion of overall functionality of the implementation in software of the \( \Psi \) and the DEMO processor. This overview applies also to the more advanced DEMO processor. The purposes of this implementation are:

\textit{i.}) Proof of concept for a production version of the DEMO processor.
\textit{ii.}) Verification of the \( \Psi \) processor specifications.
\textit{iii.}) A software tools for \( \Psi \) model construction, simulation and validation.

Appendix II.1 Overview of the current \( \Psi \) processor

The first implementation of the \( \Psi \) processor is shown in fig. 9.1.

\textbf{Figure 9.1.} \( \Psi \) processor overview.
Appendix II. \( \Psi \) Processor Implementation

The Twark Software Framework

The Twark framework is a proprietary software development environment for object oriented systems, written by the author. Twark is written in C++ . Twark supports runtime evolving information systems, is model driven, which made the software implementation much easier than the later implementation in C# for professional applications.

The Workbench

The workbench is a software module and GUI\textsuperscript{108} to control the \( \Psi \) processor. Elementary functions are:

\begin{itemize}
  \item \textbf{Init()} \quad \text{Initializes the workbench.}
  \item \textbf{Parse()} \quad \text{Parse a DMOL model file, checks grammar correctness.}
  \item \textbf{Build()} \quad \text{Build a DEMO model by linking and initializing components from the DEMO component library.}
  \item \textbf{GenDMOLmodel()} \quad \text{Renders a DMOL representation of a DEMO model.}
  \item \textbf{SaveDMOLfile()} \quad \text{Saves the current DMOL file to disc.}
\end{itemize}

\( \Psi \) model operations

The \( \Psi \) model is subject to interactions and operations. There are two modes of interaction with the \( \Psi \) model, \textit{Edit model} and \textit{Execute model}. The Edit model functions include:

\begin{itemize}
  \item \textbf{Navigate()} \quad \text{Navigate trough the \( \Psi \) model tree, display current transaction.}
  \item \textbf{Edit()} \quad \text{Read and edit transaction and actor parameters.}
  \item \textbf{New()} \quad \text{Link a new transaction and actor to an existing transaction.}
  \item \textbf{Delete()} \quad \text{Delete a transaction (tree) from its parent transaction.}
  \item \textbf{WriteTree()} \quad \text{Renders a ASCII or xml specification (Appendix I).}
\end{itemize}

The Execute Model Function

The Execute() mode or simulation mode verifies correctness of the \( \Psi \) model, calculates the current state of the \( \Psi \) model and pending Actor Query functions (Actor Interfacing Q()); the allowed state changes for this model in this state. When an Actor responds to a Query (an act) by selecting one of the available option(s) (resulting in a fact), the state of the

\textsuperscript{108} Graphical User Interface, provides the communication between the engine and the users or actors, abstracted from the actual implementation.
current model is recalculated including new Actor Interfacing functions $Q()$. The “Logging Data” function logs application of any state transition axioms and additional data for verification (Appendix I).

Switching of Operation Mode
At any time the execution can be interrupted; subject to edit model operations where the current model under investigation can be modified using the $\text{LinkPfact}()$ and $\text{DelPfact}()$ functions. $\text{LinkPfact}()$ and $\text{DelPfact}()$ functions require the default transaction state. Model construction, destruction and execution can be executed at any time.

Complete Model Introspection
At any time during simulation, the model can be inspected and various outputs can be rendered.

Output rendering
The rendered output currently available for the EO processor allows reconstruction of the ATD (Actor – Transaction Diagram) and the Process Model.

Appendix II.2 Example NLDMOL $\Psi$ model specifications

$\Psi$ model validation by non-DEMO experts requires an easier to read ASCII readable NLDMOL (Natural Language Demo Modeling language) specifications from the $\Psi$ models. The current transaction state information is included.

From this NLDMOL specification, the Actor Transaction Diagram and the Process State Diagram can be drawn without ambiguity. The currently pending Actor Queries are specified too.

It is easier for non-DEMO experts to validate these model specifications. The action Model rules and calculations of the State Model specifications are easy to include. This specification is a high quality foundation of an IT system specification.

These samples are rendered from the $\Psi$ model of fig. 8.3 (Appendix I).
The first section is a specification of all Actors.
The second section is a specification of all Transactions.
The third section is transaction decomposition according to the Composition Axiom. For each transaction it is states which transactions are required to be performed prior and from which Executor.

The fourth section specifies all transactions and their Cfact state.

DEMO model 'DEMOApplication' 'Test version DEMO Tree'.

Declaration of Actors.
Root Actor-Initiator 'A00' 'Root Initiator';
Actor 'A01';
Actor 'A10';
Actor 'A20';
Actor 'A30';
Actor 'A21';
Actor 'A31';
Actor 'A41';

Declaration of Transactions.
Transaction 'Pfact-T00';
Transaction 'Pfact-T10';
Transaction 'Pfact-T20';
Transaction 'Pfact-T30';
Transaction 'Pfact-T21';
Transaction 'Pfact-T31';
Transaction 'Pfact-T41';

Declaration of Transaction aggregation.
Actor-Initiator 'A00' 'Root Initiator'
  requires from:
    Actor-Executor 'A01' performing 'Pfact-T00'
      requires from:
        Actor-Executor 'A10' performing 'Pfact-T10'
          requires from:
            Actor-Executor 'A20' performing 'Pfact-T20'
              requires from:
                Actor-Executor 'A30' performing 'Pfact-T30'
                  end of require;
            Actor-Executor 'A21' performing 'Pfact-T21'
              requires from:
                Actor-Executor 'A31' performing 'Pfact-T31'
                  requires from:
                    Actor-Executor 'A41' performing 'Pfact-T41'
                      end of require;
        end of require;
  end of require;
end of require;
end of require;
end of require.

Declaration of Transaction states.
Actor-Initiator 'A00' 'Root Initiator' for Transaction 'Pfact-T00' has Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A01'.
Query Actor-Initiator 'Request Yes|No'.

Actor-Initiator 'A01' for Transaction 'Pfact-T10' has Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A10'.

Actor-Initiator 'A10' for Transaction 'Pfact-T20' has Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A20'.

Actor-Initiator 'A20' for Transaction 'Pfact-T30' has Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A30'.

Actor-Initiator 'A10' for Transaction 'Pfact-T21' has Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A21'.

Actor-Initiator 'A21' for Transaction 'Pfact-T31' has Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A31'.

Actor-Initiator 'A31' for Transaction 'Pfact-T41' has Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A41'.

After a number of transaction steps the following example Ψ model specification can be rendered:
Appendix II. $\Psi$ Processor Implementation

DEMOL model 'DEMOApplication' 'Test version DEMO Tree'.

Declaration of Actors.
Root Actor-Initiator 'A00' 'Root Initiator';
Actor 'A01';
Actor 'A10';
Actor 'A20';
Actor 'A30';
Actor 'A21';
Actor 'A31';
Actor 'A41';

Declaration of Transactions.
Transaction 'Pfact-T00';
Transaction 'Pfact-T10';
Transaction 'Pfact-T20';
Transaction 'Pfact-T30';
Transaction 'Pfact-T21';
Transaction 'Pfact-T31';
Transaction 'Pfact-T41';

Declaration of Transaction aggregation.
Actor-Initiator 'A00' 'Root Initiator'
  requires from:
    Actor-Executor 'A01' performing 'Pfact-T00'
    requires from:
      Actor-Executor 'A10' performing 'Pfact-T10'
      requires from:
        Actor-Executor 'A20' performing 'Pfact-T20'
        requires from:
          Actor-Executor 'A30' performing 'Pfact-T30'
          end of require;
        Actor-Executor 'A21' performing 'Pfact-T21'
        requires from:
          Actor-Executor 'A31' performing 'Pfact-T31'
          requires from:
            Actor-Executor 'A41' performing 'Pfact-T41'
            end of require;
      end of require;
    end of require;
  end of require;
end of require;

Declaration of Transaction states.
Actor-Initiator 'A00' 'Root Initiator' for Transaction 'Pfact-T00' has Cfact state of:
  Requested = false; Promised = true; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A01' .
Query Actor-Executor 'State - Acknowledge'.

Actor-Initiator 'A01' for Transaction 'Pfact-T10' has Cfact state of:
  Requested = false; Promised = false; Declined = true; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A10' .
Query Actor-Initiator 'Request | Quit'.

Actor-Initiator 'A10' for Transaction 'Pfact-T20' has Cfact state of:
  Requested = false; Promised = false; Declined = true; Quited = true;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A20' .
Query Actor-Executor 'Quit - Acknowledge'.

Actor-Initiator 'A20' for Transaction 'Pfact-T30' has Cfact state of:
  Requested = false; Promised = false; Declined = false; Quited = false;
  Stated = false; Rejected = false; Stopped = false; Accepted = false;
with Actor-Executor 'A30' .
Query Actor-Executor 'Quit - Acknowledge'.

In this representation the pending Q() query functions are shown. This representation allows verification of the allowed states of section 3.3.12.
This model is derived directly from the model in DMOL representation, which is also easy to verify:

```xml
<?xml version="1.0" encoding="windows-1252" ?>
<DMApp>DEMOApplication
  <PfactChilds>
    <Pfact>Pfact-T00
      <PfactPublic
        prmPfactIdentifier = "Pfact-T00"
        prmPfactLabel = ""
      ></PfactPublic>
    </Pfact>
    <CFact>
      <CFactPrivate
        prmRequested = "false"
       prmPromised = "true"
        prmStated = "false"
        prmAccepted = "false"
        prmDeclined = "false"
        prmQuited = "false"
        prmRejected = "false"
        prmStopped = "false"
      ></CFactPrivate>
      <CFactPublic
        prmCFactIdentifier = "Cfact-T00"
        prmCFactLabel = ""
      ></CFactPublic>
    </CFact>
  </PfactChilds>
  <ActExec>
    <Act>
      <ActPublic
        prmActIdentifier = "A01"
        prmActLabel = ""
      ></ActPublic>
    </Act>
  </ActExec>
  <ActInit>
    <Act>
      <ActPublic
        prmActIdentifier = "A00"
        prmActLabel = "Root Initiator"
      ></ActPublic>
    </Act>
  </ActInit>
  <PfactChilds>
    <Pfact>Pfact-T10
      <PfactPublic
        prmPfactIdentifier = "Pfact-T10"
        prmPfactLabel = ""
      ></PfactPublic>
    </Pfact>
  </PfactChilds>
</DMApp>
```
<CFact>
  <CFactPrivate>
    prmRequested = "false"  prmPromised = "false"  prmStated = "false"
    prmAccepted = "false"
    prmDeclined = "true"  prmQuited = "false"  prmRejected = "false"
    prmStopped = "false"
  </CFactPrivate>
  <CFactPublic>
    prmCFactIdentifier = "Cfact-T10"
    prmCFactLabel = ""
  </CFactPublic>
</CFact>
<ActExec>
<Act>
  <ActPublic>
    prmActIdentifier = "A10"
    prmActLabel = ""
  </ActPublic>
</Act>
</ActExec>
<PfactChilds>
<Pfact>Pfact-T20
  <PfactPublic>
    prmPfactIdentifier = "Pfact-T20"
    prmPfactLabel = ""
  </PfactPublic>
</Pfact>
<CFact>
  <CFactPrivate>
    prmRequested = "false"  prmPromised = "false"  prmStated = "false"
    prmAccepted = "false"
    prmDeclined = "true"  prmQuited = "true"  prmRejected = "false"
    prmStopped = "false"
  </CFactPrivate>
  <CFactPublic>
    prmCFactIdentifier = "Cfact-T20"
    prmCFactLabel = ""
  </CFactPublic>
</CFact>
<ActExec>
<Act>
  <ActPublic>
    prmActIdentifier = "A20"
    prmActLabel = ""
  </ActPublic>
</Act>
</ActExec>
Appendix II. \( \Psi \) Processor Implementation

\[
\begin{align*}
&\text{<PfactChilds>}
&\text{<Pfact>Pfact-T30}
&\text{<PfactPublic}
&\quad \text{prmPfactIdentifier = "Pfact-T30"}
&\quad \text{prmPfactLabel = ""}
&\text{</PfactPublic>}
&\text{<CFact>}
&\quad \text{<CFactPrivate}
&\quad \quad \text{prmRequested = "false"}
&\quad \quad \text{prmPromised = "false"}
&\quad \quad \text{prmStated = "false"}
&\quad \quad \text{prmAccepted = "false"}
&\quad \quad \text{prmDeclined = "false"}
&\quad \quad \text{prmQuited = "false"}
&\quad \quad \text{prmRejected = "false"}
&\quad \quad \text{prmStopped = "false"}
&\text{</CFactPrivate>}
&\quad \text{<CFactPublic}
&\quad \quad \text{prmCFactIdentifier = "CFact-T30"}
&\quad \quad \text{prmCFactLabel = ""}
&\text{</CFactPublic>}
&\text{</CFact>}
&\text{</ActExec>}
&\text{<Act>}
&\quad \text{<ActPublic}
&\quad \quad \text{prmActIdentifier = "A30"}
&\quad \quad \text{prmActLabel = ""}
&\text{</ActPublic>}
&\text{</Act>}
&\text{</ActExec>}
&\text{</Pfact>}
&\text{</PfactChilds>}
&\text{</Pfact>}
&\text{<Pfact>Pfact-T21}
&\text{<PfactPublic}
&\quad \text{prmPfactIdentifier = "Pfact-T21"}
&\quad \text{prmPfactLabel = ""}
&\text{</PfactPublic>}
&\text{<CFact>}
&\quad \text{<CFactPrivate}
&\quad \quad \text{prmRequested = "false"}
&\quad \quad \text{prmPromised = "true"}
&\quad \quad \text{prmStated = "false"}
&\quad \quad \text{prmAccepted = "false"}
&\quad \quad \text{prmDeclined = "false"}
&\quad \quad \text{prmQuited = "true"}
&\quad \quad \text{prmRejected = "false"}
&\quad \quad \text{prmStopped = "false"}
&\text{</CFactPrivate>}
&\quad \text{<CFactPublic}
&\quad \quad \text{prmCFactIdentifier = "Cfact-T21"}
&\quad \quad \text{prmCFactLabel = ""}
&\text{</CFactPublic>}
&\text{</CFact>}
\end{align*}
\]
<ActExec>
  <Act>
    <ActPublic
      prmActIdentifier = "A21"
      prmActLabel = ""
    ></ActPublic>
  </Act>
</ActExec>
<PfactChilds>
  <Pfact>Pfact-T31
    <PfactPublic
      prmPfactIdentifier = "Pfact-T31"
      prmPfactLabel = ""
    ></PfactPublic>
    <CFact>
      <CFactPrivate
        prmRequested = "false"  prmPromised = "true"  prmStated = "false"
        prmAccepted = "false"
        prmDeclined = "false"  prmQuited = "false"  prmRejected = "false"
        prmStopped = "false"
      ></CFactPrivate>
      <CFactPublic
        prmCFactIdentifier = "Cfact-T31"
        prmCFactLabel = ""
      ></CFactPublic>
    </CFact>
  </Pfact>
  <ActExec>
    <Act>
      <ActPublic
        prmActIdentifier = "A31"
        prmActLabel = ""
      ></ActPublic>
    </Act>
</ActExec>
<PfactChilds>
  <Pfact>Pfact-T41
    <PfactPublic
      prmPfactIdentifier = "Pfact-T41"
      prmPfactLabel = ""
    ></PfactPublic>
    <CFact>
      <CFactPrivate
        prmRequested = "false"  prmPromised = "false"  prmStated = "false"
        prmAccepted = "true"
        prmDeclined = "false"  prmQuited = "false"  prmRejected = "false"
        prmStopped = "false"
      ></CFactPrivate>
    </CFact>
  </Pfact>
</PfactChilds>
Appendix III. Glossary, Legends and Notations

Appendix III.1 Glossary and Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABD</td>
<td>Actor Bank Diagram</td>
</tr>
<tr>
<td>AM</td>
<td>Action Model</td>
</tr>
<tr>
<td>ATD</td>
<td>Actor Transaction Diagram</td>
</tr>
<tr>
<td>API</td>
<td>Application programmers interface</td>
</tr>
<tr>
<td>BCT</td>
<td>Bank Contents Table</td>
</tr>
<tr>
<td>C-act</td>
<td>Coordination Act</td>
</tr>
<tr>
<td>C-fact</td>
<td>Coordination Fact</td>
</tr>
<tr>
<td>Cfact</td>
<td>A software object that represents a set of coordination facts</td>
</tr>
<tr>
<td>CM</td>
<td>Construction Model</td>
</tr>
<tr>
<td>DEMO</td>
<td>Design and Engineering Methodology for Organizations</td>
</tr>
<tr>
<td>EA</td>
<td>Enterprise Architecture</td>
</tr>
<tr>
<td>EE</td>
<td>Enterprise Engineering</td>
</tr>
<tr>
<td>EO</td>
<td>Enterprise Ontology</td>
</tr>
<tr>
<td>IAM</td>
<td>Interaction Model</td>
</tr>
<tr>
<td>ISM</td>
<td>Interstriction Model</td>
</tr>
<tr>
<td>IUT</td>
<td>Information Use Table</td>
</tr>
<tr>
<td>OCD</td>
<td>Organization Construction Diagram</td>
</tr>
<tr>
<td>OFD</td>
<td>Object Fact Diagram</td>
</tr>
<tr>
<td>OPL</td>
<td>Object Property List</td>
</tr>
<tr>
<td>ORM</td>
<td>Object Role Modeling</td>
</tr>
<tr>
<td>OS</td>
<td>Object System</td>
</tr>
<tr>
<td>P-act</td>
<td>Performance Act</td>
</tr>
<tr>
<td>P-fact</td>
<td>Performance Fact</td>
</tr>
<tr>
<td>Pfact</td>
<td>A software object that represents a performance</td>
</tr>
<tr>
<td>PM</td>
<td>Process Model</td>
</tr>
<tr>
<td>PSD</td>
<td>Process Structure Diagram</td>
</tr>
<tr>
<td>PSI</td>
<td>Performance in Social Interaction, (also (\Psi))</td>
</tr>
<tr>
<td>RBAC</td>
<td>Role Based Access Control</td>
</tr>
<tr>
<td>SM</td>
<td>State Model</td>
</tr>
<tr>
<td>TAO</td>
<td>Technology, Architecture and Ontology, (also (\tau))</td>
</tr>
</tbody>
</table>
The Organization Construction Diagram, delivering the Actor Transaction Diagram (ATD) (section 2.5.1) is the most commonly used and generic DEMO Aspect Model.

It is common practice to use for external Actor roles a composite Actor role. Whether it is elementary (a single Actor) or composite (composed of multiple Actors) is considered irrelevant.

Figure III.1 Organization Construction Diagram legend.
Appendix III.2  The Operational procedure Specifications

This specification is the latest (2010) version used to represent the DEMO Action Model rules. Note that the actual implementation in the DEMO processor (section 6) is somewhat different but equivalent. Example:

when < transaction kind > of < object identifier(s) > is < transaction status$^{109}$ >
  if < Boolean logical expression >
    then < transaction kind > of < object identifiers(s) > must be < act$^{110}$ >
  else < transaction kind > of < object identifier(s) > must be < act >

Appendix III.3  The Process Structure Model

The Process Structure Model represents the current state and the allowed state transitions of transactions in detail. It is directly derived from the Organization Construction Diagram or ATD, without additional input. There are two equivalent representations, the first is shown in fig 3.17 as an example. The improved representation (2010) is shown in fig. III.2.

---

$^{109}$ Transaction state, a fact cf Φ theory (section 1.2.2), is Requested, Promised, Stated etc.

$^{110}$ Transaction Act, an act cf Φ theory (section 1.2.2), is Request, Promise, State
Appendix III.4 The State Model

The State Model (SM) as discussed in section 2.7.3 is composed of i) Object Classes, ii) Fact types, iii) Result Types and iv) Existence laws. It specifies the state space; “which facts are possible, excluding any other facts, to occur in reality in the enterprise”. The language used is WOSL [Dietz, J.L.G. 2005] and the legend is as shown in fig III.3.
The State model legends first and second part:

**Figure III.3. Legend of the Object Fact Diagram (First Part)**

The Object Classes are populated by the Objects that conform to the Object Type (see example in fig. 2.7.3). Facts that conform a Fact Type or a Result Type may occur for certain Objects of that Object Class. The existence laws add constraints to the occurrence of facts; facts whose existence exclude the existence of other facts; etc.

The State Model is expressed in the Object Fact Diagram (OFD) and the Object Property List. The Object Property List (OPL) simplifies the Object Fact Diagram. The properties are represented in a table specifying {property, domain, range}.
Figure III.4. Legend of the Object Fact Diagram (Second Part)
The OFD is based on the language World Ontology Specification Language (WOSL), developed by Dietz [Dietz 2005].

Appendix III.5 Fact Rules

Figure III.5. Legend of Fact rules diagram.
Appendix IV. The DEMO Processor in Production

The first DEMO processor application in production is a case management system for a Dutch company that delivers energy and utility services, such as water and electricity, for citizens. The basis of these services is a contract. The final production of the IT system is a tailor-made contract with the customer. The customer who signs it is also an active co-producer. The case management system is composed of the EIS (Enterprise Information System, the DEMO processor that executes DEMO models) and a closely integrated structured system for the production and management of documents, technical drawings, interfaces to legacy IT systems, including an ERP production control system. The contract contains documents originating from various sources. The contract covers issues such as type of services provided, costs, costs calculation methods, conditions for payments, instructions for the subcontractor, correspondence etc. The contract should comply with external legal regulations and internal business policies, conditions, procedures and this compliance should be enforced. The quality of the contracts should be high; errors are unacceptable, incomplete transactions, deadlocks, deadline overflow, violation of procedures etc, must be eliminated. The production of the contract is highly structured; there is a controlled sequence in the production of certain parts and there is (sometimes) an assessment and acceptance/approval step of production parts before the production may continue.

The approach to use DEMO models as foundation for IT systems is not new. In Mulder [Mulder, 2007] the SGC case has been supported by an IT system that is directly derived and developed from the DEMO models. This case study, and the other parts of this thesis as well, are a strong guidance for enterprise engineers how to develop DEMO models for complex enterprises. It shows structured analysis context diagrams, data-flow diagrams and finally the demo models.

Appendix IV.1 Project stages
The project started with DEMO modeling with the knowledgeable staff, which delivered the so-called ontological or essential model of the enterprise. Extensive shared reasoning with the staff has been done to validate that this DEMO model represents properly the service of the enterprise delivered to the customer. The ontological model is strictly implementation-independent.

The second step involved a fine-grained and recursive DEMO modeling step, starting from the ontological DEMO model, to model the detailed implementation of the production. The result is a DEMO model with 33 ontological, 'infological' and 'datalogical' transactions. Again, much effort has been devoted to this step to ensure that the implicit knowledge of the staff has been utilized. The staff is the actual and active designer of this model and
Appendix IV. The DEMO Processor in Production

accepted the final model unanimously. The resulting detailed DEMO model (ATD) delivers also high quality specifications for the document production system. The model made also clear which further design specifications were needed, for example GUI specifications.

The third step involved the implementation in executable software in a systematic way. The DEMO processor executes the DEMO model as a model-instance driven IT system. The document production system is also a model-instance-driven system.

The final step is testing and acceptance of the case management system for production.

Appendix IV.2 Practical experiences

The first two steps exploit the knowledge of key personnel and the model validation assures an optimal business-IT alignment. The application has in essence been designed by the key staff, which almost assures acceptance for full production and a good business-IT alignment.

The complex document production software has been decomposed into 33 production system software components; each software component is highly structured, well specified and very simple to implement in software (also model-instance-driven).

Each production step implemented in software can be modified with minimal impact on other production steps, which supports information system and enterprise agility.

The resources in time for the implementation in software are already less than the resources for the modeling stages. It is clear that the efforts for software implementation will be reduced further.

The acceptance testing resulted in some minor design changes of the DEMO model, an improved way of production. These modifications were very simple to implement without impact on other software modules. The staff recognized also clearly that the software implementation complies precisely with the models they designed and approved. Their general perception is that the system is very rigid in the separation of various production steps and enforces a specific way of working with little freedom. They find also that the transaction steps deviate from 'standard' GUI practice.

The communication between the actors for each production has been simplified because in a good case management system one single person does most of the contract production work. However, the precise sequence of the production steps is well guarded and there are no anomalies or incomplete transactions. Correctly executed nested transaction rollback is assured.

There is no 'workflow' or BPM modeling stage; the communication patterns between actors and the sequence of process steps are completely calculated from the DEMO model instance under execution.

The relation between project costs and required resources versus project size, measured in number of transactions, is becoming closer to linear. This is opposed to the typical exponential ratio found so far in software engineering.

The most important advantage is that the only remaining 'creative' phases are the modeling stages. The implementation in software leaves very little space for 'free and creative'
programming choices. Software engineering for adaptive case management systems is becoming more a standard production process, as in other engineering domains and as it should be.

Appendix IV.3 Applied DEMO models

This ontological DEMO model shows a part of the organization.
The detailed fine-grained DEMO model, with 33 transactions, as it is executed in production in the demo processor.
Appendix V. Curriculum Vitae

Steven Johan Herman VAN KERVEL. Born in Arnhem, Netherlands, 29 July 1950.
Education: Doctoraal examen Elektrotechniek, University of Technology, Eindhoven, Netherlands.

The research for this thesis has been carried out at Formetis BV, Boxtel, The Netherlands, and at the Delft University of Technology, The Netherlands.

Professional activities and experience
Formetis BV (1997-..). position: co-founder / partner
Formetis is a software engineering company specialized in improving the production quality and quantity of mainly financial services factories, and offers 'an implementation of your business strategy'. Formetis uses a unique 'Formula', a harmonious integration of a modeling methodology (DEMO) for business process (re)design of organizations, applies the DEMO processor and a dedicated software tools suite to build information systems.

Soltronic was a software engineering company specialized in the development of OO software tools to build information systems for hybrid high- and low-structured information such as documents, forms, business processes etc. This application development platform started in 1994 and is a (GSDP-MDE) model driven system, using an XML-equivalent model representation language. It went into production in 1997 in the banking and financial services industry.

Elektroson was an electronics engineering company, specialized in development and design of analog/digital electronics and embedded software for microprocessor control systems. Many projects were done in industrial control systems, data-acquisition systems etc. As a subcontractor we were co-developer of the first Philips CD-ROM drive (1984-1987). Elektroson developed CD-ROM (pre)mastering tools (1985-1990), a key part of the CD-ROM technology. Acquisition and project leader of an EEC Esprit R&D project with international partners (1986 - 1989). The company was sold in 1990 with 25 staff.
TNO (1977-1980)
Research at a national research institute (TNO) at the University of Nijmegen on medical ultrasonography in ophthalmology. The work included development of HF analog electronics for signal processing, mathematical modeling and computer-simulation of piezo-electronic devices, tools for calibration devices for echographic diagnostic equipment.

Technical competence
Finding engineering solutions. Extensive experience in analog and digital electronics, optical disc technology and operating systems, design of programming languages for specific applications.

Commercial competence
International experience in many countries and cultures, including Eastern Europe. Acquisition, contract negotiations and management of engineering projects. Marketing & sales of software technology in Europe, Japan and USA. Company sale & venture capital deal making.

Scientific papers have been published on the area of:
Ultrasonic diagnostic ophthalmology.
Optical disc technology (CDROM, Laservision-ROM).
Enterprise ontology, enterprise engineering, enterprise information systems.