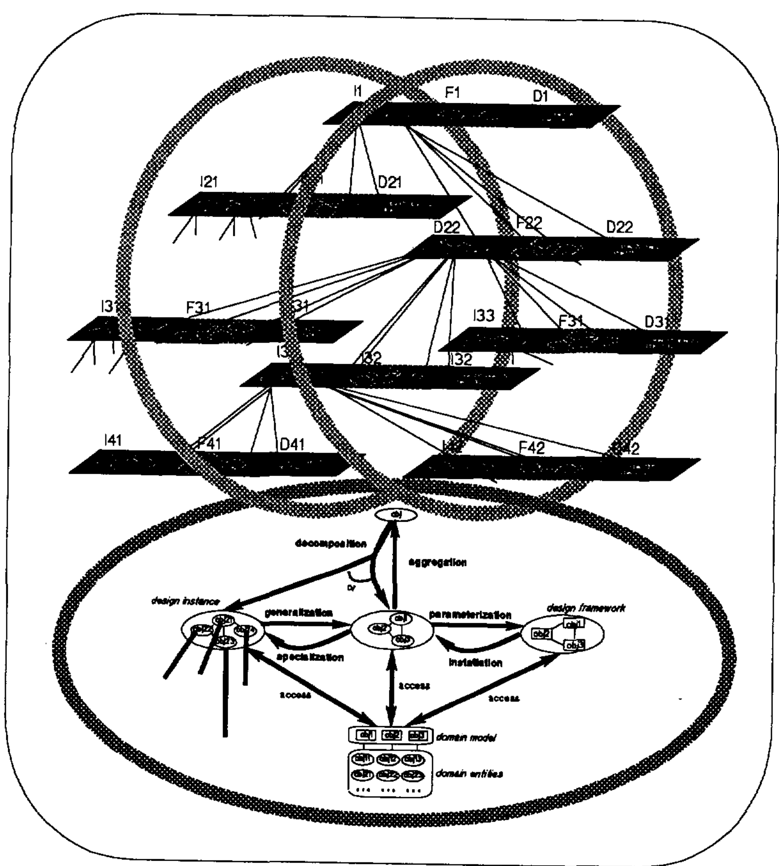
575606
3178470
TR diss 2224

TR diss
2224

RITL: An Information System for Application Reuse–in–the–Large

HaiKuan Li



RITL: An Information System for Application Reuse-in-the-Large

RITL: An Information System for Application Reuse-in-the-Large

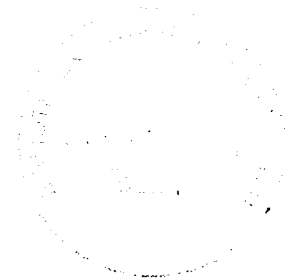
Proefschrift

Ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus,
Prof. Drs. P.A. Schenck,
in het openbaar te verdedigen
ten overstaan van een commissie
aangewezen door het College van Dekanen
op 7 juni 1993 te 14.00 uur

door

Haikuan Li

geboren te ShanXi, China



Delft University Press/1993

Dit proefschrift is goedgekeurd door de promotoren

Prof.Dr. H.G. Sol,
Prof.Dr.Ir. M. Looijen

en door de leden van de promotiecommissie:

Prof.Drs. C. Bron,
Prof.Dr. W. Gerhardt,
Prof.Dr. H. Koppelaar,
Prof.Dr. D.J. McConalogue

Published by:
Delft University Press
Stevinweg 1
2628 CN Delft
The Netherlands

ISBN 90-6275-871-1

Copyright c 1993 by Haikuan Li. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocoping, recording or otherwise, without the prior written permission of the author.

Stellingen

behorende bij het proefschrift

RITL: An Information System for Application Reuse-in-the-Large

door

HaiKuan Li

1. The necessity of maintenance is reflected in Hinds' law in programming: any given program, when running, is obsolete, and if a program is useful, it will have to be changed.

2. The real difficulty of maintaining information systems does not lie in the complexity of dealing with the existing systems, but lies in the endless ambitions to have a better system.

3. It is ironic that very much information fails to be represented in system development (forward engineering), whereas quite similar information has to be recovered in reverse engineering for the purpose of maintenance. That is, system producers make puzzles and let reverse engineers solve them.

4. It is too late to look at maintenance of information systems after delivery.

5. The retrieval of system components from a component library is a major problem for component reuse. Such a problem can be solved in reuse-in-the-large by organizing the components according to the process of system development and maintenance, i.e. by putting them at the 'places' where they are created.

6. Hashing is not only a means to mess up something but also a means to sort out the mess.

7. A sketch is not a painting, but a guideline for imagination. The charm of a sketch is allowing appreciators to derive their own images respectively, so as to enjoy the creativeness of both the author and the appreciators themselves.

8. If scientific research can be divided into several stages, then the first stage is a dream, and the last stage is another dream.

To Meiling and Yun

Contents

Preface v

1 From Reuse-in-the-Small to Reuse-in-the-Large 1

1.1 A survey of application reuse . . . 1
1.1.1 What is application reuse? . . . 2
1.1.2 Why is application reuse necessary? . . . 3
1.1.3 How does application reuse happen? . . . 4
1.1.4 The development of application reuse . . . 8
1.1.5 Summary . . . 9
1.2 The outline of the research . . . 10
1.2.1 Research setting . . . 10
1.2.2 Research questions . . . 12
1.2.3 The contents of the dissertation . . . 14

2 The Reuse of Abstract Datatypes and Algorithm Structures 17

2.1 Approaches to component reuse . . . 18
2.2 Basic considerations . . . 20
2.3 Our approach . . . 22
2.3.1 Principles for components representation. . . . 22
2.3.2 Outline of a reuse support system . . . 23
2.3.3 Ocomp and Scomp: Semantics . . . 25
2.3.4 Ocomp and Scomp: Syntax . . . 27
2.3.5 Generation of operations . . . 29
2.4 Reuse . . . 30
2.4.1 Scomp reuse . . . 30
2.4.2 Ocomp reuse . . . 31
2.5 An example: binding to C++ . . . 31

3 The reuse of system architectures and design templates 35
3.1 Introduction 36
3.2 Actor and its refinement 37
3.2.1 The definition of an actor 37
3.2.2 The hierarchy of an actor 38
3.2.3 The rules for actor refinement 40
3.2.4 Actor and EDFG 42
3.3 System architecture 44
3.3.1 Creating a system architecture 44
3.3.2 Reusing system architecture 45
3.4 Templates of system design 47
3.4.1 Template 48
3.4.2 Import and inheritance 51
3.5 Summary 54

4 The reuse of large-scale components 55
4.1 The abstractions for component modeling 56
4.2 The pragmatic model of large-scale components 58
4.3 The operating model of large-scale components 61
4.3.1 The relationship between an object and its refinement . 62
4.3.2 A design framework: a parameterized design 63
4.3.3 Domain resources: objects for instantiation 63
4.3.4 Design instance: a specialized design 63
4.3.5 The formalization 65
4.4 Management of large-scale components 66
4.4.1 Retrieval problem 67
4.4.2 Semantic data base and semantic data modeling. 67
4.4.3 The representation of large-scale components 68
4.4.4 The component base 71
4.4.5 The process of creating and reusing large-scale components 74
4.5 The capability of reusing large-scale component 77
4.5.1 The life-cycle oriented reuse 77
4.5.2 Incremental domain analysis 79
4.6 An information system for application reuse-in-the-large 81
4.7 Summary 83

5 The Specification of Large-Scale Components 85

5.1 The specification of large-scale components 85

5.1.1 Multiple-level Specification 86

5.1.2 Specifying large-scale components 90

5.2 An example of large-scale components 94

5.3 The consistency verification 100

5.3.1 The concepts of actor and *edfg* 101

5.3.2 The refinement consistency 103

5.4 Summary 116

6 Reuse-in-the-Large and Application Prototyping 117

6.1 Introduction 117

6.2 Representing large-scale component in terms of EDFG 118

6.2.1 The concepts about EDFG 118

6.2.2 The process of EDFG decomposition 119

6.2.3 The definition of prototypes in terms of EDFG 120

6.2.4 Representing large-scale components in terms of prototypes 121

6.3 Using large-scale components in application prototyping 124

6.3.1 The process of application prototyping 124

6.3.2 Component specification and EDFG mapping 127

6.3.3 The management of large-scale components 129

6.4 Summary 131

7 Reuse-in-the-large: A case study 133

7.1 Introduction 133

7.2 Representing a large-scale component 137

7.2.1 Specifying the C-REPORTER (top node) 138

7.2.2 Specifying the PRINTER (A second level node) 144

7.2.3 Specifying the RETRIEVER (Another second level node) 149

7.3 Browsing and manipulation 154

7.3.1 Browsing a large-scale component 154

7.3.2 Manipulating a large-scale component 160

7.4 Generating alternative design 163

7.5 Generating implementation 166

7.6 Summary 170

8 Epilogue **173**
8.1 Reuse-in-the-large 173
8.2 Supporting management of information system 174
8.3 Supporting dynamic modelling of applications 176

Summary **179**

A Scomps and Ocomps in Duplex Shell **185**

B Index **213**

C Bibliography **219**

D Curriculum Vitae **231**

Preface

All things in the world are originated from existence. Existence is originated from non-existence.

– Lao Tsu, 6th century B.C.

This dissertation provides system developers with a set of methods for application reuse, particularly for application reuse-in-the-large. The idea of reuse-in-the-large, in contrast with that of traditional reuse (the reuse of small code components), aims at reusing large-grain components of information systems and, more importantly, the information about the creation of the components. Such kind of reuse reflects the requirements for developing large and complex applications. The methods for application reuse-in-the-large, as provided in this dissertation, include a method for reusing abstract data types and algorithm structures, a method for reusing system architectures and design templates, and a method for modelling, specification, management and manipulation of large-scale components. These methods are applied in an information system for application development and maintenance with reuse-in-the-large.

This dissertation would have never been possible without the help of many people. Acknowledgements are due to Prof.Dr. H.G. Sol, my promoter, for his constructive and imaginative way of directing research, for his continuing inspiration and for his challenge to my intellectual capability, and for his careful review and effective revision of this dissertation.

I am greatly indebted to Prof.Dr.Ir. M. Looijen, my second promoter, for his careful reading of the drafts, for his fruitful guidance and confidence in the dissertation work, and for his continuing encouragement and support.

Acknowledgements are also due Prof.Dr.Ir. J. van Katwijk for his valuable contribution to the dissertation work leading to several publications, for the

unforgettable cooperation with him for several years, and for his advise and help.

I would like to thank Prof.Drs. C. Bron of Groningen University for his continually keeping an eye on the dissertation work, and for his critical comments and constructive suggestions on the earlier version of the dissertation.

Special thanks are given to Ir. F. Ververs and Drs. E. M. Dusink for a good cooperation with them on the same project during a long period of time, for many beneficial discussions with them, and for their help in every possible way concerning the research related to the dissertation.

I appreciate very much the efforts given by Prof.Dr.Ir. F.W. Jansen, Prof.Dr. S.C. van Westrhenen and Prof.Dr.Ir. v.d. Poel, for their understanding and support of the dissertation work. I am grateful to Dr. W. Tracz of IBM Corporation, USA, for his enthusiastic approval of the approach regarding the dissertation, when he was the chairman of WISR'91.

I also wish to express my gratitude to Prof. L. Latour of the University of Maine, USA, Prof. L. Luqi of Navy Graduate School, USA, Prof. M. Wirsing of Passau University, Germany, Prof. R. Mittermeir of Klagenfurt University, Austria, Prof. H. Weber of Dortmund University, Germany, and Dr. R. Prieto-Diaz of Software Productivity Consortium, USA, for their support and encouragement.

I am very grateful to Mrs G.M.A. Stoute for her correction of almost all my papers. I am also very grateful to Mr. A.W.W.M. Biegstraaten and Mr. H.M.M. Engelen who were always ready for quickly solving problems with the computer systems and for helping me in various ways.

Chapter 1

From Reuse-in-the-Small to Reuse-in-the-Large

The drive to create reusable rather than transitory artifacts has aesthetic and intellectual as well as economic motivations and is part of man's desire for immortality. It distinguishes man from other creatures and civilized from primitive societies.

– Peter Wegner, 1984

In this chapter a survey on the state of the art in the field of application reuse is provided, and our research on application reuse is introduced.

1.1 A survey of application reuse

The discrepancy between the demand for large and complex information systems and the ability to build such systems is still large. For a long time many efforts aiming at solving the problem have been tried from several aspects including system modelling methodologies, programming languages, development environments, and application reuse. Although all the efforts are useful and capable of providing partial solutions of the problem, application reuse is especially attractive to improve system quality and productivity, because a large amount of expensive work has been found redundant in the practice of system development [41], and quite some achievements to avoid the redundancy are already there[116].

In this section, we describe what application reuse is, why it is necessary, how it can be achieved, what the current achievements are, and our future work.

1.1.1 What is application reuse?

Reuse happens in our daily life as well as in various scientific areas and is, therefore, a general concept. We see construction workers using the same kind of bricks to build different kinds of buildings, architects applying the same design principles to different architectural designs, mathematicians using the same formulas to solve different problems, and physicists using the same laws to explain different phenomena. If we derive the concept of reuse from the general experience of daily life and various scientific areas, then *reuse* is just the reapplication of the same object or the same knowledge to different applications.

Similarly, *application reuse* is the reapplication of *artifacts* and/or *knowledge* on the development of one *application* to another application, in order to reduce the effort of application development and maintenance of that other application [18]. Artifacts refer to any products of the engineering process of application development, such as systems, components and documentation. The *knowledge* refers to the facts and experience known by system developers and maintainers about system development and maintenance, such as the information on system analysis and design. An application is a system which provides a set of services for solving some type of user problem[96]. An application can also be a part of an *information system* which is produced for the functional use of the information system [81]. An information system is a system for collecting, storing and processing of data. It is the sum of data sets, application software, system software, hardware, procedures (among which working instructions), documentation (among which manuals), technical features and people.

Application reuse can be classified into *reuse-in-the-small* and *reuse-in-the-large*. The former is centered on the reuse of small *system components*, and the latter is centered on the reuse of large-grain components and *design information*, i.e. the knowledge about the creation of the components. A *component* or *system component* in this dissertation, particularly refers to one of the parts that make up an application, which may be software, hardware

or other artifacts, and may be subdivided into other components.

Although there is quite some experience on reuse-in-the-small and it is proved to be moderately successful in some areas [38][116][95] [50], such kind of reuse is insufficient to improve the process of *system development* and *maintenance* [18]. In contrast, reuse-in-the-large aims at obtaining design information from each phase of system development, and at taking this information as a basis to improve the process of similar development and maintenance[9][93] [106]. System development refers to the activities of *analysis, design*, and *implementation* [20]. Analysis is a process of studying the capability needed by a user to solve a problem or achieve an objective. According to Sol, design is a process of problem solving, which can be viewed as a chain of transformations in which the products of various activities in the process superimpose in successive layers[127][112]. The activities include the definition of the system architecture, components, data, interfaces for a system to satisfy specified requirements[51]. Implementation is a process to realize a design in more concrete terms; in particular, in terms of hardware, software, or both[96]. Moreover, system maintenance refers to the activities of modifying a system or component after delivery to correct faults, improve performance or other attributes or adapt to a changed environment[96][81].

1.1.2 Why is application reuse necessary?

According to Cox[35], the software crisis causes software to be too expensive, its quality insufficient and its development almost impossible to be managed. Our opinion is that application reuse, especially the reuse of components of applications, might reduce the pressure of the software crisis.

First, it was reported that the Electronics Industries Association (US) estimated that the DOD (The Department of Defence) would spend $23.1 billion on mission-critical software in one year[38]. Another report described that the cost of maintenance can be reduced by 90%, using application reuse [116], while according to [95] the cost of software maintenance is usually about 75% of the total cost of the whole life-cycle of software. Other studies [50] show that on a given project 40% of a design and 75% of the code are reused. If we assume that the expenses needed for application reuse are considerably small, the cost of system development and maintenance can be significantly reduced by applying reuse.

Thirdly, application reuse may lead to a better process of system development. In this area some achievements have already been made while others are in progress[18]. On the one hand, we have already means to manage source code components, such as libraries[100][20][26], on the other hand, reverse engineering and re-engineering, as discussed later on, provide improved means for managing system development and maintenance, and the development of some environments is in progress [17] [9]. Moreover, there are good examples of managing *design information*, e.g. the Draco system[93], in which the knowledge about the system development process itself is kept under control.

1.1.3 How does application reuse happen?

The question how application reuse happens basically depends on what kind of application reuse is applied. Our discussion will be focused on reuse-in-the-small and reuse-in-the-large.

Reuse-in-the-small

Reuse-in-the-small is centered on reusing *small* system components. A system component is *small* if its implementation or description can be compared with a primitive language construct such as a class in C++, a package in Ada and so on[18], or it can be represented in terms of 10-100's lines of source code[16]. The research on reuse-in-the-small is basically concerned with *component representation* and *component management.*

Component representation deals with the form and content of system components. The research on component representation is often oriented towards programming languages. Most methods to achieve appropriate representations of components are based on classical techniques: divide and rule [24] and abstraction. A large application is usually built up from a set of building blocks, e.g. modules in Modula 2 (dividing); secondly, applications may be generated from domain knowledge, e.g. the rules and the patterns for application construction [122] (ruling); and, thirdly, a complex component can be represented as a black box and, therefore, can be reused at an abstract level, e.g. the specification of an Ada package (abstraction).

Component management is concerned with *repositories*, *classification* and the *retrieval* of system components. A component repository is a component

library providing permanent , archival storage for components and relevant documentation. Component classification is a mapping of a collection of components to a taxonomy or the process of determining such a mapping[96]. Component retrieval is the activity to search a required component from a component library. One kind of component management, perhaps the most successful one at this time, is based on the application of data base techniques to the management of system components. Using data base techniques, library components can be classified and retrieved according to the properties of the components[100][88].

Programming language facilities form another basis for managing components. *Inheritance* is a prime example of this kind. By organizing system components in a hierarchical structure, the content of an existing component can be shared by its descendants[125].

The problems with reuse-in-the-small are well known[18][124][43]: First, reuse-in-the-small provides insufficient means and artifacts to manage and support the process of system development and maintenance. Secondly, building up a system from small code components leaves a lot of work to be done in building the *architecture* that bind the components into a whole system, while the cost to build this architecture, as indicated by Biggerstaff[18], is typically very much larger than the savings afforded by reusing a set of small components. An *architecture* refers to the organizational structure of a system or component, i.e. the structure and relationship among the components of a system[96][51].

Reuse-in-the-large

In contrast with reuse-in-the-small, reuse-in-the-large is concerned with reusing large-grain components and, perhaps more important, with the reuse of design information. A component is said to be a *large-grain* component if it consists of a set of small system components or their composition, such as subsystems or even complete systems.

The requirements for reuse-in-the-large are quite different from that of reuse-in-the-small. While it is often reasonable to regard a small code component as a reusable black box, this view is hardly possible for a large-grain component. A large-grain component — say a subsystem — usually contains much more features concerned with a particular application than a small com-

ponent. When such a component is reused in a new application, these features must be replaced by the features matching the requirements of the new application. If the large component is a parameterless black box, it is hardly possible to adapt the component to different applications. If it is parameterized on these features, reusers shrink at the sight of too many parameters[64]. On the other hand, it is nearly impossible to ask a reuser to understand the inside structure of a large-grain component in sufficient detail either.

However, reuse-in-the-large is possible. The development of reuse-in-the-large can be traced back to the original motivation of system maintenance. The goal of maintenance, according to Looijen[81], is far beyond the correction of the design mistakes in a system. It also aims at adapting an existing system to the application environment which is continually changing, at improving the performance or other attributes of a system, at extending the functionality of a system, and at enabling a system to defend itself from possible turbulence, i.e. improving robustness. These activities lead to several research aspects including corrective maintenance, adaptive maintenance, perfective maintenance, additive maintenance, and preventive maintenance[81]. The activities of maintenance result in successive versions of evolving systems, which, according to Biggerstaff, is highly focused reuse-in-the-large[18]. Moreover, research[125] pointed out that such kind of reuse appears to be a more important source of increased productivity than reuse of small code components in different applications.

In our opinion, maintenance is a kind of system development with reuse-in-the-large. A support for such an idea is Basili's reuse-model. Treating maintenance as a reuse-oriented development process, according to Basili[9], provides a choice of maintenance approaches and improves the overall evolution process.

Some other technologies on reuse-in-the-large are *reverse engineering* and *re-engineering.*

Reverse engineering is defined by Chikofsky and Cross II, as " the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction"[31]. Methods supporting reverse engineering are concerned with providing means for a better understanding of an existing system, identifying reusable artifacts in the system and recovering the artifacts

about the creation of the system, such as the design specification of the system. The Desire System[17] is an example of reverse engineering, aiming at recovery of domain knowledge, external information and other useful information about systems.

Re-engineering aims at evolving old systems or building new system out of existing ones by utilizing the information provided by reverse engineering. Methods supporting re-engineering may be concerned with an evolutionary process of development or the reconstruction of existing systems and the information on the development of the systems. Basili's approach [9] is also an example of re-engineering, which provides a possibility of reusing the information on requirements, design, coding and testing in the development process.

Unfortunately, reverse engineering and even re-engineering almost always have to be applied to poorly designed and incomplete products without a structure appropriately supporting reusability [94], i.e. the capability of being reused. A way to avoid this problem is to create reusability during system development, i.e. to improve the skills in *forward engineering.*

Forward engineering is the process of going from requirements through design to implementation[31]. Forward engineering providing support for reuse is called *reuse-supported forward engineering.* Methods supporting such forward engineering may be concerned with formalizing the information of analysis and design, with generalizing components, and with building high abstractions of components. Some application generators[125] are examples of tools supporting reuse-supported forward engineering. Such tools usually keep up rules and domain knowledge and provide possible automation at some phases of analysis, design, implementation and even maintenance [93] [15] [125].

Although reverse engineering, re-engineering and reuse-supported forward engineering provide experience that forms a basis for dealing with reuse-in-the-large, there is still quite a large research space to look into. Biggerstaff[18] even claims that reuse-in-the-large introduces a whole new set of research problems. These problems especially include how to represent, manage and manipulate both large-grain components and the design information, so as to allow reuse over a broad range of target systems and over the process of system development and maintenance.

1.1.4 The development of application reuse

Reuse-in-the-small and reuse-in-the-large are rooted in *programming-in-the-small* and *programming-in-the-large* respectively.

Programming-in-the-small and reuse-in-the-small

System development lived through its childhood until the mid of 1960s when research attention was concentrated on how to represent structure and symbolic information and the elementary understanding of control flows[108]. In the mid 1960s, according to Shaw, research attention on system development was shifted from its childhood to programming-in-the-small when the abstractions of algorithms and data structures emerged, and the studies on program specification were started[108]. However, programming-in-the-small is program centered. The programs are usually simple and made through individual effort; the system components are code components; and the component specifications are often simple input/output specifications[108].

The history of application reuse may be traced back to the component reuse which, as early as 1950, according to Wegner, when Wilkes, Wheeler and Gill recognized the importance of subprogram libraries of reusable programs[125]. In the mid 1960's, the capability of component reuse was further improved when data structures emerged[59][108]. In the late 1960's, the idea of component reuse was formulated by McIlroy who advocated what he called a component manufacturing facility [84][50]. However, corresponding to programming-in-the-small, component reuse was reuse-in-the-small at that time. The characteristic is that the reused components are usually small code components such as procedures, functions, classes and so on. Some techniques applied to reuse-in-the-small include parameterization, information hiding and inheritance. The idea is building libraries which are well populated with reusable code components.

However, many reuse problems were not solved yet before research attention was shifted from programming-in-the-small to programming-in-the-large, such as how to identify reusable components, how to retrieve the components from the library, how to fit a component into different applications and so on.

Programming-in-the-large and reuse-in-the-large

In the mid 1970s, according to Shaw[108], research attention on system development was shifted from programming-in-the-small to programming-in-the-large. *Programming-in-the-large* is centered on the development of complex systems which are implemented by a team rather than by an individual. It is recognized that system development may include several phases such as analysis, design, implementation and maintenance. The artifacts produced during the development include both the components of code and design information.

Programming-in-the-large is quite different from programming-in-the-small. First, the coding which was a major part of programming-in-the-small became less important to programming-in-the-large. The cost of coding may be only 13-18% of the whole cost of system development[50] [32]. Secondly, maintenance has become a very important part of the system life-cycle. The cost of system maintenance may reach 75% of the whole cost of a system during its lifetime[36]. Thirdly, the importance of system design was increased. The cost of design together with analysis is 32-46% of the whole cost of system development[50] [32]; additionally, both code components and design information are needed for maintenance.

The shift from programming-in-the-small to programming-in-the-large reflects the maturity of development techniques. However, programming-in-the-large introduces a new set of reuse problems, such as maintenance, evolution, prototyping, and re-engineering. Attracted by the importance of these problems, research attention on reuse was shifted from reuse-in-the-small to reuse-in-the-large.

The characterization of the shift from reuse-in-the-small to reuse-in-the-large is the shift from the reuse of small components to the reuse of both small and large components, from the reuse of code components to the reuse of both code components and design information, and the shift from aiming at improving implementation to aiming at improving the process of system development and maintenance.

1.1.5 Summary

In this section, a survey and a classification of application reuse were provided. Application reuse does provide technologies to deal with the reusabil-

ity of system components and the knowledge about system development and maintenance. The classification of application reuse was provided in terms of reuse-in-the-small and reuse-in-the-large. The former is centered on the reuse of small source code components, whereas the latter is centered on the reuse of large-grain components extended with design information.

Comparing reuse-in-the-large with reuse-in-the-small, we conclude that — although there is quite some experience on reuse-in-the-small and it proved to be moderately successful in some areas — such kind of reuse is insufficient to improve the process of system development and maintenance. In contrast, reuse-in-the-large aims at obtaining design information from each phase of system development, and taking this information as a basis to improve the process of system development and maintenance.

Future developments in the field of application reuse should continually concentrate on methods and support for representation and management of the artifacts to be reused, especially for the representation of large-grain component and design information.

1.2 The outline of the research

In this section we describe the research setting, research questions and the contents of the dissertation.

1.2.1 Research setting

As discussed in the previous section, reverse engineering and re-engineering are important ways of application reuse. However, they lack efficiency in achieving reuse-in-the-large if they are applied to poor representations of applications [94][124][106]. Unfortunately, as pointed out by Webster[124], conventional system development methodologies are product-oriented; they were developed to address the artifacts being designed, they do not adequately address design information, particularly the representation about the information for the creation of the artifacts. Accordingly, we need to address novel requirements for system development.

The novel requirements for system development, according to Sol[113], are the need for a *common frame* of reference, concerning a coherent set of descriptive building blocks, the need for an *extendible* system description and

analysis context, the need for an *iterative* process of analysis and synthesis, and the need for the generation of *alternative* possibilities.

To meet the requirements above, Sol introduced the concept of dynamic modelling. The philosophy of dynamic modelling, he wrote, aims at the improvement of staff performance by creating an environment which supports problem solving processes. The modelling method should stress the specification of conceptual and empirical models for better understanding of existing situations and possible alternative solutions. The modelling should be based on possible changes in task structures and working processes, and on an effective application of technology.

As to the state of the research on dynamic modelling, a start has been made by Dur and Bots[25]; many description methods are based on static modelling, adding dynamic aspects at later stage. Sol indicated that we need to seek other effective coordination means, instead of spending too much efforts to decide which description method promotes the translation into software best. The essence of dynamic (system) modelling requires more detailed investigation into proper concepts and ways of representation.

In this dissertation we develop an approach for application reuse-in-the-large. Our research is focused on: (i) the representation of large-grain components extended with design information, (ii) the realization of a *reuse infrastructure* for the acquisition, storing and manipulation of the artifacts to be reused, and (iii) the possibility of generating alternative applications in an *application domain.*

A reuse infrastructure refers to the information and its structure which must be available to system developers, together with the auxiliary information needed to locate and manipulate this information[3]. An application domain refers to a set of current and future applications marketed by a set of common capabilities and data [96].

From a reuse point of view, this approach integrates reuse into the process of system development and maintenance. It is a systematic way of creating and reusing artifacts and is capable of improving the process of system development and maintenance. However, the information to be handled, as focused on by us in this approach, is limited to the information which can be represented in terms of popular notations such as diagrammatic notations and formal specification[72]. We do not try to handle further upstream de-

sign information[124], which is usually informal and concerned with the early aspects of a system development process.

The process of system development and maintenance corresponds to a life-cycle model, in particular an improved waterfall model. A life-cycle model is a description of the process of system development and maintenance according to the life-cycle of systems to be developed. A system *life-cycle* refers to the period of time that starts when the system is conceived and ends when the system is no longer available for use[51]. A *model* is a representation of a process, phenomenon or concept in the real world[51].

The *waterfall model*[125] is a traditional life-cycle model. According to this model, the development of system proceeds through a number of stages: *analysis* → *design* → *implementation* → *maintenance*[125]. The improved waterfall model is similar to the life-cycle model in object-oriented design[20], which allows a jump in waterfall model from each higher stage to any lower stage and a trace-back from each lower stage to any higher stage [20]. The former reflects the need to reuse lower level resources at an early stage of system development, the latter reflects the need of iterative analysis, design, implementation and maintenance.

1.2.2 Research questions

Many methodologies have been proposed as being effective to application reuse[125][18]. But we claim that none of them provides a representation form to meet the requirements of reuse-in-the-large in both the scope and the organizational structure of the information to be reused, and none of them supports a systematic process to be followed.

There are several well-known approaches that have been proposed for reusing design information and/or large-grain components, including MILs[99], Draco[92] and EPROS[48]. MILs (Module Interconnection Languages) provide a description of system structure and resources flowing among system components[99]. MILs are useful for reusing system architecture and can be used to support system design and analysis. The research question is, however, according to Prieto-Diaz[99], how could an MIL be expanded or argumented to include information about the availability of resources and modules? More information is needed to indicate the specification of such modules and resources. Draco is the first approach that supports application reuse in terms

of *domain analysis*. Domain analysis refers to the process of identifying, collecting, organizing, analyzing, and representing a domain model and system architecture from study of existing systems, underlying theory, emerging technology, and development histories within the domain of interest[96]. Draco is successful for identifying reusable artifacts prior to developing the first system in an application domain and organizing them together for application reuse, but the cost of domain analysis, as evaluated by Balda[7], makes the Draco approach very expensive in its initial use. EPROS supports reuse by generating prototypes from a formal description of a system, although it does not provide means to extend the reuse process into domain analysis.

In order to support application reuse-in-the-large a new approach is required to support several features missing in the discussed approaches. Our goal is to seek an approach to support representation, management and reapplication of reusable artifacts, and to support a systematic process for application or system development and maintenance with reuse-in-the-large.

We need to go one step further than the related approaches. First, in contrast with MILs, an approach for application reuse-in-the-large may not only allow the reuse of system architectures (system structures in MILs) but also provides the *specification* of resources, particularly concerning what the necessary resources may be, where they are placed and how they are reused. Secondly, in contrast to the Draco approach, we need an *incremental domain analysis*: The artifacts which are acquired during the process of developing the earlier system(s) may be both reused and enriched during the process of developing other systems later on. The enrichment can be continued until domain analysis is completed. Thirdly, in order to support maintenance and reuse large-grain components rather than frequently compose them from primitive components, we need to maintain an instance of a system and its design. This will also allow system developer and customer to deal with their requirements over an existing design or an executable system, and to develop new systems based on the modification and reconstruction of an existing one in a problem domain.

Consequently, we intend to address an *information system* which supports (application) system development and maintenance with Reuse-in-the-Large. It supports the reuse of both large-grain components and design information, supplies artifacts to be reused at the time and place when reuse is necessary,

and provides the capability of acquiring artifacts to be reused during the process of reuse.

1.2.3 The contents of the dissertation

Our research on application reuse can be traced back to 1988, when our attention was mainly focused on the reuse of source code components. At the very beginning, our work was centered on the reuse of C and C++ program units[69]. Later on, a method was addressed for the reuse of algorithm structures and abstract data types [68], the former representing the context to compose a program, and the latter representing the lower level entities to be composed with. Such a method led to a proposal for a reuse supporting environment[70][71]. From such a method we noticed that (1) the context for component composition as represented in terms of algorithm structures may play an important role in application reuse, (2) component modelling may not only be useful to package operations oriented to certain objects, but also to package the operations with respect to certain algorithm structures. For a detailed discussion, see also chapter 2.

Soon after the method above was addressed, we understood that the context for component composition should be an architectural structure for large-grain components. Therefore, our research attention was shifted to the EDFG approach [65] in mid-1990, which allowed us to look into the possibility of reusing system design information. During this period, a method iwaswass presented to reuse both source code components and design information, including system architectures and other artifacts of system design. According to this method, a design can be represented in terms of Extended Data-Flow Graphs (EDFGs) and formal specifications. The graphs, specifications and code components can be organized into a hierarchical structure, representing different levels of abstraction. Such a structure can easily be maintained and reconstructed to meet varieties of design targets. For a detailed discussion, see also chapter 3.

As soon as we had experience in reusing large-grain components and design information concerning a specific project, we intended to go one step further by introducing a general method for reuse-in-the-large. Such a method concerns the modelling, representation, management and reapplication of large-scale components. A *large-scale component* is an extended system component which

contains design information, including concept, content and context about the construction of varieties of large-grain components such as subsystems or even complete systems in an application domain. The concept is described by a functional specification; the content is the refinement and implementation of a component; the context is the architectural structure or algorithm to compose a component from lower resources. All information of a large-scale component is organized in a hierarchical structure, representing *different levels of abstraction.* All the information can be incrementally accumulated and reused in the process of system development and maintenance. For a detailed discussion, see also chapter 4.

An *abstraction* refers to a representation or specification of a system or component that focuses on the information relevant to a particular purpose and ignores the remainder of the information. *Different levels of abstraction* refers to a set of interrelated abstractions which are classified into several levels according to an architecture. A higher-level abstraction catches the commonalities of a set of lower level abstraction. A lower-level abstraction refines a higher-level abstraction with detailed or additional parts and properties.

In order to represent and understand large-scale components in a precise manner so as to reuse them effectively, we introduce a principle for the specification of large-scale components and describe how to apply such a principle for the representation of large-scale components. Moreover, we address a method for the consistency verification of large-scale components. This method suggests that the verification can, with a few rules, be simplified by inference from the extended data flow graphs.

As an investigation of applying the approach for reuse-in-the-large to the practice of system development, we looked into how a large-scale component is capable of supporting system prototyping. By such an example, some performance of the method was verified. For a detailed discussion, see also chapter 6.

In order to convince readers how RITL is practically applied in a more specific manner than before, we discuss how reuse-in-the-large happens in a support environment, regarding a particular application domain: Student Management System (SMS), which is represented in terms of a large-scale component. Particularly, we discuss how SMS is specified in terms of RSL, an available specification language, how it is understood and manipulated

and how varieties of designs and implementations can be generated from the large-scale component so as to meet varieties of requirements in an application domain. For a detailed discussion, see also chapter 7.

Scientific research has no end, "Still achieving, still pursuing, learn to labour and to wait (*Longfellow*)."

Chapter 2

The Reuse of Abstract Datatypes and Algorithm Structures

algorithms + data-structures = programs.

N. Wirth, Prentice Hall, 1976.

As a preparation of application reuse-in-the-large, the representation and reapplication of source code components are discussed in this chapter. When looking into some popular programming languages and some approaches of reuse[18], we find that *abstract data types* and *algorithm structures* can be a basis for building larger-grain components. While abstract data types can be used to represent objects, algorithm structures can be used to bind the objects into large-grain components. An abstract data type is a data type for which only the properties of the data and the operations to be performed on the data are specified, without concern for how the data will be represented or how the operations will be implemented[96]. An algorithm structure is a parameterized module from which one or more abstract data types are extracted and replaced by proper parameters. A module refers to a logically separable part of a system[51].

2.1 Approaches to component reuse

Quality requirements concerning the reusability of system components [85][15][69] seem difficult to be met. On the one hand, system components should be *soft* (changeable) so that they can flexibly be reused for different requirements and/or in different application environments (the *adaptability* demand); on the other hand, system components should be *hard* (unchanged) as well, so that they can form the basis for further component composition (the *composibility* demand) without modification of the components. Additionally, its reuse should be easy to be carried out (the *simplicity* demand). [1] Adaptability refers to the quality of system components which allow different application constraints and user-needs to be satisfied[51]. Composibility refers to the quality of system components from which large-components or systems can be built [104]. Simplicity refers to the quality of system components, which allows the reapplication of the components with less effort than that of creating the components. Additionally, that a component can be flexibly reused or has flexibility for reuse, according to Peterson[96], implies the ease with which the component can be tailored for the use in applications or environments other than those for which it was specifically designed.

Years of efforts from different corners have been spent in addressing reuse. The approaches to reuse can be divided into two groups depending on the nature of the components being reused, namely, *composition*-based approaches and *generation*-based approaches[15].

The composition-based approaches, according to Biggerstaff, are characterized by the fact that the reusable components are largely atomic and ideally unchanged in the course of their reuse. In generation-based approaches components are viewed as rules and patterns [15] and are meant to be modified (hopefully mechanically) in their reuse.

An example of composition-based approaches is using *abstract data types* or its extension as in object-oriented programming. The extension of using abstract data type results in object-oriented programming in which the decomposition of a system is based upon *objects* [20]. An object, according to Booch[19], is an entity whose behaviour is characterized by the operations that it suffers and that it requires from other objects. Object-oriented program-

[1] Simplicity does not imply that the components themselves should be simple, it does imply that composing systems or complex components should be easy.

ming supports a certain degree of flexibility in the definition and composition of components. The basic components in object-oriented programming are usually represented as abstract data types or its extension, e.g. classes in C++. A class denotes the potential to create a set of similar but unique objects[19]. It serves to describe the common properties of a set of objects and to specify the behavior of all instances. An abstract data type provides a way of *information hiding.* The extension of abstract data types also supports a way for component composition, typically, the mechanism of *inheritance.* Information hiding is an abstraction technique in which the visibility of defined entities of components is limited and controlled. Inheritance is a technique for hierarchical resource sharing which provides means to define components as extensions of previously defined ones[126] [85].

Examples of the generation-based approaches are the template approach [121] and the frame-based approach[10]. The template approach as advocated by Volpano[122] aims at reusing the representation of algorithm in terms of *templates.* A template is an algorithm specification which is free from any commitment to specific types and representations of data. In other words, the specification of algorithms is parameterized in terms of polymorphic abstract data types. As such, the instantiation of a template requires abstract data types as (actual) parameters. However, the template approach supports neither inheritance directly for the creation of abstract data types which are maintained in a library, nor flexible *algorithm tailoring* which seems essential to allow the usage of similar (though not equal) algorithms. Tailoring is a technique for representing components which provides support to change the components into varieties. Algorithm tailoring refers to generating a representation of a similar algorithm with reusing an existing one.

The frame-based approach, advocated by Bassett [10], aims at reusing *abstract algorithms*, which supports a flexible tailoring of algorithms. In this approach, an algorithm corresponds to a piece of program, and an abstract algorithm is defined in terms of a frame. A frame is built up as a macro with a set of macro parameters. In terms of a macro almost any part of a program appearing in a frame can be defined as a macro parameter. Apart from providing a wide form of genericity, this kind of parameterization makes both the tailoring of data structure and the tailoring of algorithm available. Genericity is a property of a system component. A system component has

genericity if it has more than one interpretation depending on parameters representing types or subcomponents[85].

We think that the frame-based approach is especially successful in those cases where a high degree of algorithm tailoring is required. It does support flexible reuse of algorithms, but it does not support abstract data types to be used for component construction.

Considering the facts above, we find that the techniques which are applied to the different approaches above can complement each other for reuse. On one hand object-oriented approaches allow the creation of abstract data types or its extensions, they fail in algorithmic reuse as effective as that of the template approach and frame-based approach. On the other hand the template approach is especially useful for the specification of algorithms but needs abstract data types to instantiate the algorithms into system components. Moreover, if a higher degree of tailoring is required, e.g. to solve problems concerned with *same-as-except* [2], it might be better to extend an object-oriented approach with a frame-based approach.

Consequently, in order to emphasize the reusability of both abstract data types and algorithms as well as the tailoring of components, we think that a single approach is needed in which elements of compositional approaches and elements of generation approaches are combined.

2.2 Basic considerations

In this section we describe our basic considerations about how to meet the quality requirements of components including *adaptability, composibility* and *simplicity* as mentioned at the beginning of section 1, and how to enhance the reusability.

Firstly, considering the techniques used in the approaches that were discussed in the last section, we think that *inheritance, information hiding, genericity* and *tailoring* are especially useful techniques to meet the quality requirements of system components, which are described as follows:

[2] *same-as-except* means that components to be reused do not exactly match requirements of reuse. This may be considered to be a general possible notion of reusability. Refer to Bassett[10] for further detail.

- *genericity and tailoring* allows components to be used in several applications and therefore support *adaptability.*

- *inheritance* provides a systematic method for component composition and therefore supports *composibility*;

- *information-hiding* deals with the complexity of data and algorithms, and therefore supports *simplicity*.

Secondly, considering the components used in the previously discussed approach, we believe that *reusability* can be enhanced by using more than a single kind of component, which are complementary to each other. These components may include

- *abstract data types and objects*, which support both information hiding and inheritance as appearing in object-oriented approaches and provide values for the instantiation of algorithm structure.

- *algorithm structures* which, as appearing in e.g. the template approach, support genericity and tailoring, and provide means to compose large components or systems from abstract data types or objects.

Reusability refers to the degree to which *resources* can be used in more than one application or in building other components. Resource refers to any entity placed in a library for the purpose of reuse. Finally, we also think that reusability can be enhanced by addressing the following problems:

- *The contradiction between information hiding and algorithm reuse:* if information hiding is supposed to hide the implementation of components, the algorithm structures which are contained in the implementation and might be reusable, are hidden as well then. However, if the implementation is open, there would be no more information hidden.

- *The contradiction between different approaches:* a component representation, which is useful in composition approaches[15], may not be a proper representation in generation-based approaches. The ideal components being reused in composition-based approaches are atomic and

unchangeable ones, which may be *use-as-is* [3]; while the components being reused in generation-based approaches are often patterns (patterns of code and patterns of transformation rules) woven into the fabric of generator programs [15].

Our approach is based on the hypothesis above, which will be discussed in the next section.

2.3 Our approach

In this section we propose some principles for component representation and we explain their influence on reuse.

2.3.1 Principles for component representation.

We propose principles for component representations: one is object-oriented abstraction, and the other is structure-oriented abstraction.

- *Object-oriented abstraction* is a principle for component representation which catches the commonality of operations which are able to operate on a type of object.
- *Structure-oriented abstraction* is a principle for component representation which catches the commonality of the operations whose implementations are isomorphic with respect to the algorithm structures used.

Object-oriented abstraction resembles what has been represented in object-oriented approaches; structure-oriented abstraction is (partially) what has been represented in both the template-based approach and the frame-based approach as mentioned earlier.

According to the principles above, we propose two kinds of components: object-oriented components and structure-oriented components.

- An *object-oriented component (Ocomp)* is a component which represents an abstract data type or its extension, such as a class.

[3] *use-as-is* means to use something atomically, which is considered to be the simplest possible notion of reusability[10].

- A *structure-oriented component (Scomp)* is a component which represents a set of interrelated algorithm structures and a set of operations with respect to the algorithm structures.

According to the definition of abstract data type algorithm structure, an *operation* can be considered to be the sum of abstract data types and algorithm structures. An operation is a program unit such as a procedure or function. Our approach supports representing an operation as two separate parts: one representing algorithm structures and the other representing abstract data types, i.e. Ocomps and an Scomp. In this case the reusability of a program can be enhanced: the Ocomps may also serve as parameters of (other) Scomps, while, conversely, the Scomp may receive other Ocomps as its parameters and, in this way, new programs can be produced. In section 2.3.3 and 2.3.4, we will discuss the semantics and the syntax of the two components in further detail.

Limitations. Although our proposed approach is supposed to reflect the characteristics of both compositional and generational approaches, the characteristics are limited to inheritance, genericity, information hiding and tailoring.

2.3.2 Outline of a reuse support system

A system which supports component reuse, realizing our ideas, can be implemented as a reuse support environment. This environment supports programmers to represent algorithm structures and abstract data types or classes using Scomps and Ocomps respectively. It also permits the elements of an Ocomp to be parameters for the instantiation of an Scomp and the result of the instantiation, conversely, can be a part of other Ocomps.

The system contains a *component manager* and an *operation generator*. The component manager logically manages two component bases, an *Obase*, and an *Sbase*, corresponding to the two kinds of components. The operation generator is responsible for the transformation of algorithm structures and abstract data types into new operations that can be used to compose new Ocomps. The component manager can be a data base management system. The Obase contains a set of Ocomps, the Sbase contains a set of Scomps.

Below we diagrammatically represent a reuse support environment in terms of three blocks and the relationships between them.

The block *Ocomp* depicts an Ocomp base which contains a set of abstract data types (ADTs). The block *Scomp* depicts an Scomp base which contains a set of algorithm structures; *frame(ADTs)* is an abbreviation of algorithm structures, *frame* indicates a representation of algorithm structure; *(ADTs)* indicates the arguments of the frame. The construct implies that the algorithm structures can be instantiated with abstract data types. For the layout of *frame(ADTs)*, see also the description of semantics and the syntax of Scomps in this section. The block in the middle is a set of operations which are candidate elements of new Ocomps.

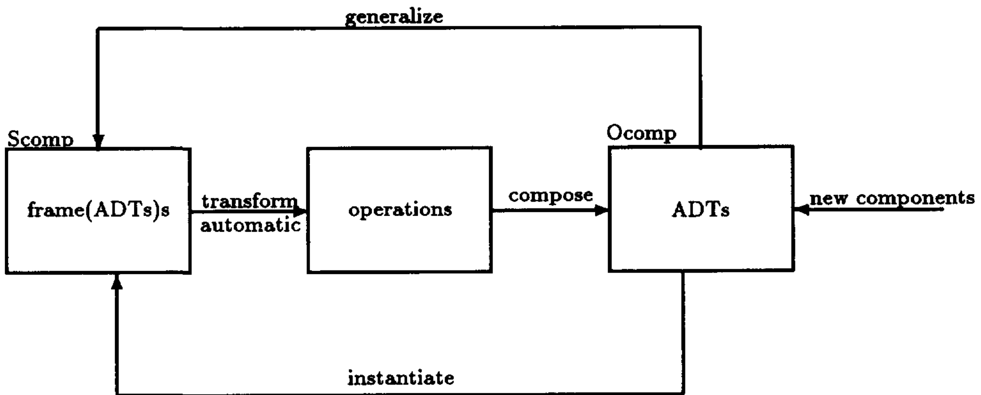


Figure 2.1. A reuse support system.

Let's explain how the system works. Programmers produce *new components* in the form of Ocomps. An Ocomp defines an abstract data type in terms of data structures, variables and operations. The implementation of the operations, such as procedures, functions and so forth, can be generalized by the programmer into *frame(ADTs)s*, from which Scomps can be produced. An Scomp defines an algorithm structure in terms of a template[10]. An Scomp may also contain a set of descriptions about its instantiation. The algorithm structures (indicatively *frame(ADTs)*) can be instantiated by using a set of other *ADTs*. If the Scomp already contains a description of such an instantiation, this instantiation can be done *automatic*ally by a *transform*er. The result

of the transformation is a set of new operations. These new operations can then be used to *compose* other new Ocomps. The programming complexity is, therefore, reduced.

2.3.3 Ocomp and Scomp: Semantics

In this section we discuss the semantics of Ocomp and Scomp. In principle an Ocomp is not a new language construct. It can be either a *module* in Modula, a *class* in C++, or a *package* in Ada. However, for a convenient discussion, we provide an independent semantics and syntax in contrast with that of an Scomp.

Ocomp An Ocomp is defined by two tuples: one providing specification, the other, implementation. The implementation defines data types and provides the operations which operate on values of the data types; the specification is an abstraction of the implementation. The representation of an Ocomp is as follows:

Ocomp(Ocomp-spec, Ocomp-impl)

Ocomp-spec(Oname, Data-spec, Ooperation-spec, Inherit-spec)

Ocomp-impl(Oname, datatype-def, Ooperation-def)

In the *Ocomp-spec*, the *Oname* is an identifier; *Data-spec* is a set of data types which are defined in the Ocomp; *Ooperation-spec* is a set of signatures (headings) of operations; *Inherit-spec* (inheritance specification) is a set of names of existing Ocomps, whose elements are inherited by the Ocomp being defined.

In the *Ocomp-impl*, the *Oname* is equal to the *Oname* of the corresponding *Ocomp-spec*; *datatype-def* is a set of data type definitions and a group of variable declarations; *Ooperation-def* is the implementations of a set of operations.

In the *Ooperation-spec*, we distinguish between two cases: the operation specification (a signature of operation) without an accompanying Scomp's name specifies what we call a *normal operation* (a normal operation is an operation which is implemented in the Ocomp), whereas a specification accompanied by an Scomp's name specifies what we call a *virtual operation*,

such an operation can be generated from combining the Ocomp under consideration with the Scomp the name is provided for.

Scomp An Scomp is also defined by two tuples: one providing specification, the other one, implementation. The specification specifies a group of interrelated algorithm structures, a set of operations (to be implemented by instantiating the algorithm structures), and one or more names of Ocomps that define abstract data types for the instantiation of the algorithm structures. The implementation of the Scomp consists of one or more representations of algorithm structures and a set of instantiation descriptions. Each of these instantiation descriptions can be used to generate one of the operations which are specified in the specification.

An Scomp can be described as:

Scomp(Scomp-spec, Scomp-impl)

Scomp-spec(Sname, Algorithm-spec, Soperation-spec, Srelation-spec)

Scomp-impl(Sname, Algorithm-structs, Instant-desc)

In the *Scomp-spec* the *Sname* is an identifier, *Algorithm-spec* is a set of signatures of algorithm structures contained in the Scomp, and *Soperation-spec* is a set of signatures of operations. *Srelation-spec* is a specification of the relationship between the Scomp and some Ocomps which are capable of providing parameters (abstract data types) for instantiating the algorithm structures contained in the Scomp.

In the *Scomp-impl*, the *Sname* is equal to the *Sname* in *Scomp-spec* ; *Algorithm-structs* is a set of representations of algorithm structures; and *Instant-desc* is a set of instantiation descriptions.

The algorithm structures are represented in terms of frames. The *frames are a form of macros*[103] [10]. We prefer the frame as a basis for the representation, since a frame is capable of representing an algorithm structure and is capable of being tailored into a set of alternative operations.

The *instant-desc* is a set of instantiation description. An instantiation description is a group of instructions that tell a generator how to conduct parameter substitutions, and how to modify the algorithm structures (if slightly different algorithm structures are required). Concrete techniques for operation

generation and for such a tailoring are described by e.g. Bassett[10].

2.3.4 Ocomp and Scomp: Syntax

The syntax of the Ocomp can be briefly presented as shown in figure 2.2.

Ocomp Specification	**Ocomp Implementation**
Ocomp Oname **interit** Ocomps {	**Obody** Oname {
data:	**data:**
type$_{11}$ variables$_{11}$;	[*data-struct*$_1$];
type$_{12}$ variables$_{12}$;	[*data-struct*$_2$];
...	...
type$_{1m}$ variables$_{1m}$;	[*data-struct*$_w$];
operation:	type$_1$ variables$_1$
output$_{11}$ op$_{11}$(input$_{11}$);	type$_2$ variables$_2$
output$_{12}$ op$_{12}$(input$_{12}$);	...
...	type$_r$ variables$_r$
output$_{1n}$ op$_{1n}$(input$_{1n}$);	**operation:**
output$_{21}$ op$_{21}$(input$_{21}$):Scomp$_1$;	[*prog-unit*$_{11}$];
output$_{22}$ op$_{22}$(input$_{22}$):Scomp$_2$;	[*prog-unit*$_{12}$];
...	...
output$_{2k}$ op$_{2k}$(input$_{2k}$):Scomp$_k$;	[*prog-unit*$_{1n}$];
}	}

Figure 2.2. The syntax of Ocomps.

The left part of figure 2.2 "**Ocomp ...interit...{ data:... operation:...}**" provides a syntactic framework of an Ocomp specification:

- "*Oname*" is the name of the Ocomp, an identifier;
- "*Ocomps*" is a list of the Ocomp names, a set of identifiers;
- "*type*$_{ij}$ *variables*$_{ij}$" is a variable declaration which consists of a data type and a set of variables. The former is an identifier, the latter is a list of identifiers;
- "*output*$_{ij}$ *op*$_{ij}$ *(inputs*$_{ij}$*)*" is an operation signature. *output*$_{ij}$ is the output type of the operation, an identifier; *op*$_{ij}$ is the name of the operation, an identifier; the *inputs*$_{ij}$ is a set of arguments, a set of identifiers;

- "*output$_{ij}$ op$_{ij}$(inputs$_{ij}$):Scomp$_i$*" is an operation signature with a post-fix—a name of an Scomp, an identifier.

In the right part of figure 2.2 "**Obody ... { data: ... operation: ... }**" provides a syntactic framework of an Ocomp implementation.

- "*[data-struct$_i$]*" is a definition of data structure;

- "*type$_i$ variables$_i$*" is a variable declaration;

- "*[prog-unit$_{ij}$]*" is a program unit, such as procedure, function, etc.

In addition, the variables specified in the specification are different from the variables specified in the implementation. Both of them can be accessed by the specified operations, but only the variables which appear in the specification can be accessed by other Ocomps.

The syntax of Scomp is briefly presented as shown in figure 2.3.

Scomp Specification	**Scomp Implementation**
Scomp Sname{	**Sbody** Sname {
algorithm:	**algorithm:**
alg$_1$ {*type*:args$_{11}$;*oper*:args$_{12}$;*expr*:args$_{13}$}	alg$_1$ [*frame$_1$*];
alg$_2$ {*type*:args$_{21}$;*oper*:args$_{22}$;*expr*:args$_{23}$}	alg$_2$ [*frame$_2$*];
...	...
alg$_h$ {*types*:args$_{h1}$;*oper*:args$_{h2}$;*expr*:args$_{h3}$}	alg$_h$ [*frame$_h$*];
operation:	**instance:**
output$_1$ op$_1$(input$_1$):Ocomps$_1$;	op$_1$ [*macrodefs$_1$*];
output$_2$ op$_2$(input$_2$):Ocomps$_2$;	op$_2$ [*macrodefs$_2$*];
...	...
output$_r$ op$_r$(input$_r$):Ocomps$_r$;	op$_r$ [*macrodefs$_r$*];
}	}

Figure 2.3. The syntax of Scomps.

In the left part of figure 2.3 "**Scomp ... { algorithm: ... operation: ... }**" provides a syntactic structure for Scomp specification.

- "alg_i{type:$args_{i1}$;oper:$args_{i2}$;expr:$args_{i3}$} " is a signature of an algorithm structure. alg_i is the name of the structure, an identifier; *type, oper and expr* are keywords; $args_{ij}$ is a list of macro arguments, a set of capitalized identifiers.

- "*output*$_i$ op_i*(input*$_i$*):Ocomps*$_i$" is an operation signature with a postfix. *output*$_i$ is an output type of the operation, an identifier; op_i is the name of the operation, an identifier; *input*$_i$ is a parameter list, a set of identifiers; *Ocomps*$_i$*;* is a list of Ocomp names, a set of identifiers.

The right part of figure 2.3 " **Sbody ... { algorithm: ... instant ... }** " provides syntactic structure for a Scomp implementation.

- "alg_i *[frame*$_i$*]*" includes the name of an algorithm structure and its representation. alg_i is the name, an identifier; *[frame*$_i$*]* is the representation: a program unit in macro form[10].

- "op_i *[*macrodefs$_i$*]*" is an instantiation description. op_i is the name of the description, an identifier; *[macrodefs*$_i$] is a set of expressions for macro substitution.

2.3.5 Generation of operations

In the approach proposed, the system contains an operations generator. The function of the operations generator is to transform a virtual operation into its implementation.

The input of the operations generator are: (1) an instantiation description represented by the *macrodefs*$_i$, *macrodefs*$_i$ $\in$ *Instant-desc*; (2) algorithm structures represented by the *frames* , *frames* $\subseteq$ *Algorithm-structs*; and (3) the *Ocomps*$_i$, *Ocomps*$_i$ $\subseteq$ *Srelation-spec*. The output of the operations generator is an operation implementation.

The specification of a virtual operation in an Ocomp specification includes two elements: a signature of the operation, and the name of an Scomp. The signature tells a user how to use the operation; the name of an Scomp provides information to the system about which Scomp is to be used in generating the implementation of the operation. The Scomp (its specification and implementation) contains all the input required for the generation. For a detailed discussion, see also [70].

2.4 Reuse

By using Scomps and Ocomps, both abstract data types and algorithm structures can be a basis for component reuse. The Ocomp supports inheritance by specifying names of other Ocomps. The Scomp supports genericity by presenting algorithm structure. Tailoring can be conducted by the instantiation description, and information hiding is implemented by specification. Even more important: an Ocomp can be used as a basis for Scomp construction and vice versa. Some other features about reuse are discussed in the following sections.

2.4.1 Scomp reuse

Reuse of Scomps can be categorized into *foreseen* reuse and *unforeseen* reuse. Foreseen reuse focuses on reusing operations, unforeseen reuse is focused on reusing algorithm structures.

Foreseen reuse It is quite common for a programmer to make decisions about the level of genericity of program fragments, based on some expectations of the use of such a unit, and for the reuser to make decisions on actual parameters. Nevertheless, too many or too complex parameters often confuse reusers.

The Scomp structure enables programmers to provide reusers not only with algorithm structures, but also with groups of operations which are loosely-isomorphic with respect to the algorithm structures and are considered as thesauri. The operations being included in the tuple of *Soperation-spec* (see 2.3.3), show the programmer's expectation on reuse, and which is, therefore, called *foreseen reuse*.

Unforeseen reuse Reuse that was not expected by the programmer is called *unforeseen reuse*. As we know, in foreseen reuse the programmer has expectations on reusing the algorithm structures and, therefore, specifies a set of operations originating from the algorithm structures in Scomps. Providing the specification, however, depends on the programmer's experience (hence the name foreseen reuse). Complementary to the foreseen reuse, in order to support unforeseen reuse, the Scomps allow reusers to create new applications

of the algorithm structures by tailoring.

Such tailoring is supported by allowing the augmentation of new operations to an Scomp, and the modification of the algorithm structures. Both the augmentation and modification can be safely conducted on the original algorithm structure.

Since each operation originates independently from the algorithm structures, the augmentation of a new operation to an Scomp has no side effect upon other operations.

By using macro and default techniques, such as proposed by e.g. Bassett[10], tailoring the algorithm structures for a new operation has no side effect on alternative reuse of the algorithm structures.

It might be a hard job for reusers to directly reuse the algorithm structures, because they have to read it. However, the result of the reuse can be a contribution to other reusers, i.e. the newly specified operations can directly be reused from now on. The reusability of the modified Scomp is, therefore, increased.

2.4.2 Ocomp reuse

Reuse of Ocomps can be categorized into two elements. First, the facilities of an existing Ocomp can be inherited by a new Ocomp. Although a new Ocomp may augment or replace any subsets of the inherited data and operations, the ideal case is *use-as-is*. This reflects the character of composition-based approaches (section 2.1). Secondly, the facilities of existing Ocomps can be used to instantiate algorithm structures of Scomps. New operations can, therefore, be produced by means of the operations generator (section 2.3.5), which reflects the character of generation-based approaches (section 2.1).

2.5 An example: binding to C++

In order to explain a possible binding with C++, we give an example with definitions of an Scomp and an Ocomp. Figure 2.4 is a specification of an Scomp, while figure 2.5 is the implementation of this Scomp.

```
Scomp Scan {
    algorithm:
      typedef ELEM * TEXT;
      scan{ type: TEXT, ELEM;
            oper: END, MATCH;
            expr: OP}
    operation:
      int lr-scan(STRING, int, char);
      int rl-scan(STRING, int, char);
      int list-lr-scan(LIST, int, RECORD):Obj2;
      int list-rl-scan(LIST, int, RECORD):Obj2;
      int find-lr (DOCUM, int, CARD):Obj3;
      int find-rl (DOCUM, int, CARD):Obj3;
}
```

Figure 2.4. An Scomp Specification.

In the specification there is an algorithm structure, *Scan*, and six operations. The *Obj2, Obj3*, are names of existing Ocomps respectively.

In the implementation (**Sbody**), under **algorithm**, there is a frame (a representation of an algorithm structure) in C++, in which the capitalized terms TEXT, ELEM, END and MATCH are *macro variables (Mvariables)*; under **instance**, there are several instantiation descriptions beginning with an identifier. There is a special instantiation description beginning with an identifier *default*, in which all undefined Mvariables have values assigned in the description. The other instantiation descriptions begin with an identifier which is the name of the description.

Following each identifier, there is a set of *macro expressions (Mexpressions)* with expression {<*Mvariable*>={<*value*>}}. This expression implies substitution of the value for the Mvariable [Bassett-87]. The value can be a term or expression of the programming language (C++).

```
Sbody Scan {
    algorithm:
      scan(TEXT t, int p, ELEM e) {
          int j;
          j=p;
          while(! (END(j) || MATCH(t[j],e)))
              j=j OP 1;
          if(END(j))
              return(NIL);
          else
              return(j);
      }
    instance:
      default: {TEXT={STRING}} {ELEM={char}}
              {MATCH={eq}} {OP={+}} {END={end}}

      lr-scan:
      rl-scan: {OP={-}}
      list-lr-scan: {MATCH={match}} {TEXT={LIST}}
              {ELEM={RECORD}}
      list-rl-scan: {*list-lr-scan, rl-scan}
      find-lr: {TEXT={DOCUM}} {MATCH={find}}
              {ELEM={CARD}}
      find-rl: {* find-lr, rl-scan}
}
```

Figure 2.5. An Scomp Implementation.

There are also some special expressions with expression {* <identifier> ...}, which provide a hint for reusing the previous Mexpressions identified by the *identifier*. This kind of inheritance is only for reducing the duplication of Mexpressions.

In order to illustrate how Scomps can be applied to construct object-oriented components, *InfProc*, an Ocomp, is represented in figure 2.6 and 2.7.

The Ocomp, *InfProc*, inherits three other existing Ocomps, Obj2, ObjL,

and ObjX; it contains three variables and six operations. There are three virtual operations which refer to two existing Scomps: *Scan* and *ShellSort*. In the *Obody*, two data types and three operations are defined in C++. The "..." stands for source code being omitted.

```
Ocomp InfProc interit Obj2, ObjectL,OBjX {
    data:
        RESULT x,y;
        DATE z;
    operation:
        BOOL retrieve(DATE);
        void evaluate();
        int list-lr-scan (LIST, int, RECORD):Scan;
        int list-rl-scan (LIST, int, RECORD):Scan;
        void ListSort(LIST,int):ShellSort;
        void show-result();
}
```

Figure 2.6. An Ocomp Specification.

```
Obody InfProc {
    data:
        typedef struct { ... } RESULT;
        typedef struct { ... } DATE;
        RECORD a,b;
        RESULT c;
    operation:
        BOOL retrieve(DATE){ ... }
        void evaluate(){ ... }
        void show-result() {...}
}
```

Figure 2.7. An Ocomp Implementation.

Chapter 3

The reuse of system architectures and design templates

Hardware engineering's levels of integration are a good model for software engineering. "Object-oriented" means different things at different levels of integration.

Brad J. Cox, IEEE Software 1990

Although the combination of abstract data types and algorithm structures is useful for system construction, such an effort is limited to software coding. In this chapter a method is presented for the reuse of system components and *system designs* which includes *system architectures* and *design templates.* A system design refers to the abstractions and mechanisms that provide the required behavior of a system or a component. A system architecture refers to the organizational structure of a system or a component. A design template refers to a pattern of design representation. According to this method, a system design can be represented in terms of Extended Data-Flow Graphs (EDFGs), textual specifications and the templates of such specifications. The graphs, specifications and code components can be organized into a hierarchical structure, representing different levels of abstraction. Such a structure can easily be maintained and reconstructed to meet varieties of requirements.

3.1 Introduction

The method presented in this chapter is concerned with the *EDFG approach* [65] in a system development environment *SEPDS*[66]. The EDFG approach is an approach for the design and implementation of distributed and real-time systems using Extended Data Flow Graphs (EDFGs)[65].

In the EDFG approach[66], a set of extended data flow graphs (EDFGs) are generated during the process of system design. An EDFG is the representation and specification of a coherent set of actors and the relationships between them. An *actor* in the EDFG approach is a system component which can be used to represent a process, subsystem or some other system component. An actor can be refined into a lower-level EDFG. According to this approach, the process of system design starts by creating a highly abstract EDFG and by following different levels of refinement until the actors contained in the EDFGs are found in a component base, or the grain of the actors are suited to be implemented in terms of small code components. The refinement of EDFG is to provide a detailed design of at least one actor of the EDFG to be refined with a subgraph (sub-EDFG) [65].

The method advocated is intended to enhance the reusability of system design. The reusability of system design refers to the reapplication of the representations of one system or component to the construction of similar ones in an application domain. We think that the reusability can be enhanced if the system designs are explicitly represented and can easily be understood and manipulated (modified and reconstructed) towards a variety of target systems.

An EDFG is a diagrammatic notation. Although such a notation may be criticized by its lack of precise semantics, it can be effectively applied to the practice of system development because of its graphical nature; a semantic extension of this notation can be provided in terms of a textual specification. We think that the method advocated is to a large degree independent of the kind of development method and design representation. As such, EDFGs can be replaced by other, perhaps more precisely defined notations.

The remaining part of this chapter is organized as follows: in section 3.2 actors and EDFGs are introduced; in section 3.3 the representation and reapplication of system architectures are discussed; in section 3.4 we describe some mechanisms for actor composition and parameterization; in section 4. a conclusion is drawn.

3.2 The actor and its refinement

In the EDFG approach, the *actor* is the basic element of system design. An actor can be an abstraction of a subsystem or its components.

3.2.1 The definition of an actor

An actor, a, as defined in the SEPDS[66], is a 5-tuple as in figure 3.1. A **link** represents a place holder, a buffer of data values as they flow from actor to actor; and a **token** is an arbitrary complex value structure. It is possible to allow more than one token contained in a single link at the same time. A link appearing in the actor definition can also be a **state link**. A state link is used to indicate global data indicated by a global variable. An **action** refers to a system component which can be a high abstraction of a subsystem or a primary function or operation. Additionally, IFS is short for *Input Firing Set* and OFS is short for *Output Firing Set*[66], also called input links and output links in this chapter respectively.

$a = (IFS(a), OFS(a), FIRE(a), PRE(a) POST(a))$
$IFS(a) = (l_1, l_2, ...l_n)$: *a set of input links connected to the actor;*
$OFS(a) = (l_{n+1}, l_{n+2}, ...l_{n+k})$: *a set of output links connected to the actor;*
$FIRE(a) : IFS(a) \longmapsto$ *Boolean: the fire condition indicating the links that should (or should not) contain a token for the actor to fire;*
$PRE(a) : IFS(a) \longmapsto$ *Boolean: the precondition that needs to be satisfied to allow the actor to perform its action properly;*
$POST(a) : IFS(a), OFS(a) \longmapsto$ *Boolean: the postcondition resulting from firing the actor.*

Figure 3.1. The definition of actor.

actor spec a **is**
IFS(a) $= (l_1 : d_1, l_2 : d_2, ...l_n : d_n)$
OFS(a) $= (l_{n+1} : d_{n+1}, l_{n+2} : d_{n+2}, ...l_{n+k} : d_{n+k})$
FIR(a): IFS(a) $\longmapsto$ *Boolean*
PRE(a): IFS(a) $\longmapsto$ *Boolean*
POST(a): IFS(a), OFS(a) $\longmapsto$ *Boolean*
end a

Figure 3.2. The module structure of an actor.

In order to describe the tokens a link may have, the term *link type* is addressed. A link type is a description of a set of links of actors which are allowed to contain tokens of the same data type. Any link of the link type is called a member of the type.

Following the definition of an actor and the definition of a link type, the *module structure* of actors is described as shown in figure 3.2. A module structure is an abstraction, describing the structure of modules. A module is a unit of program or a design representation of a system or system component. The a appearing in this figure is the name of an actor. $l_i : d_i$ means the link l_i is a member of the link type d_i, i=1,2,...n+k.

3.2.2 The hierarchy of an actor

There are two factors concerning the representation of an actor. First, the representation should be easily generated during the process of system design. Secondly, the representation should support a kind of hierarchy for component structuring and component composition.

Normally, systems and components can be structured or composed by using two types of hierarchies, i.e. a vertical hierarchy and a horizontal hierarchy. According to Tracz[118], the former refers to levels of abstraction or stratification, and the latter refers to aggregation and inheritance. In this subsection we discuss the vertical hierarchy of actors. Such a hierarchy reflects the refinement process of system design in the EDFG approach, providing different levels of a component.

In order to represent the vertical hierarchy, we classify the actors as either *primitive* or *non-primitive*. A primitive actor can be implemented in terms of a small code component. A non-primitive actor can be represented by a combination of its abstract representation and its refinement. The former provides an integral understanding of the actor as a whole, the latter consists of a set of other actors which can be primitive or non-primitive and which are the constitution of the refinement of the actor. The hierarchy of an actor is based on the definition given by Levy[65] as follows.

definition 1 *An actor which is called non-primitive is an actor which is refined into other actors. That is, if an actor a is non-primitive, then $\exists a_1, a_2, ..., a_n$ such that*

$$a \doteq (a_1, a_2..., a_n)$$

where $\doteq$ *stands for refinement and each* a_j *is either primitive or non-primitive,* $0 \le j \le n$.

According to the definition above, we suggest that a representation of a non-primitive actor is as follows,

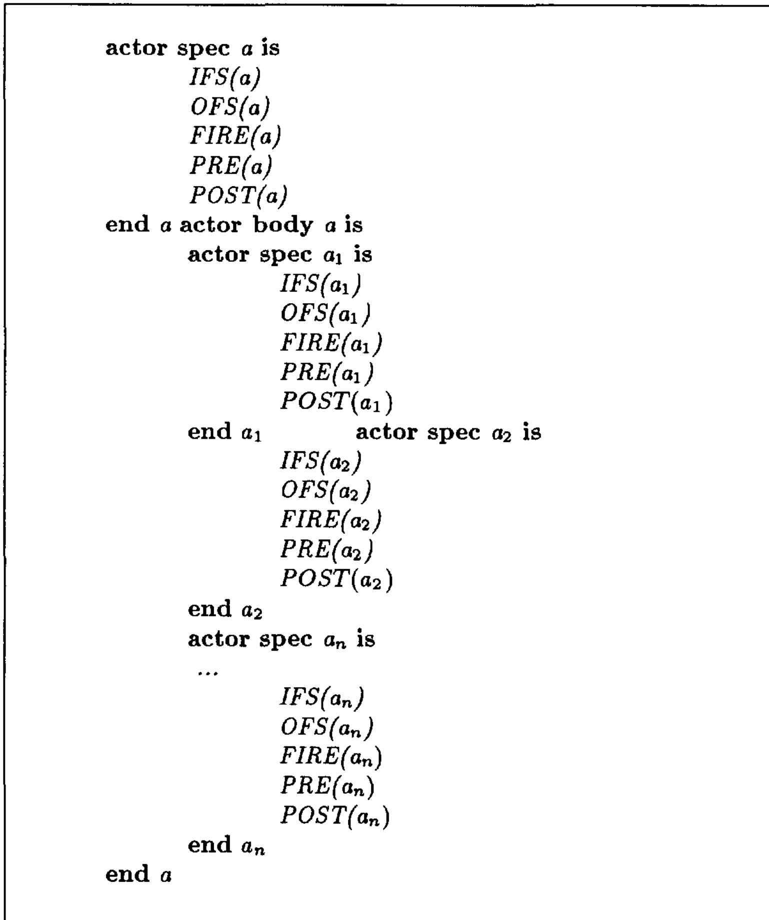


Figure 3.3. The representation of a non-primitive actor.

A non-primitive actor consists of two parts: a specification and a refinement. The specification part is identified by **actor spec** and a refinement part is identified by **actor body**. The specification part specifies a single actor which is composed by all actors appearing in its body. The refinement part packages a

set of interrelated actors which may be either non-primitive actors or primitive ones.

3.2.3 The rules for actor refinement

In the previous subsection we provided a structure to represent the actor and its refinement in terms of an actor specification and a one level refinement. Now we discuss this representation in certain details. There are several representation rules which the representation of actors must follow. Our discussion will concentrate on the representation of actors and the relationships between the specification part and the refinement part of an actor, as well as on the relationships between the actors of the refinement part.

The representation of a non-primitive actor a is

actor spec a **is**
$IFS(a)=(l_1 : d_1, l_2 : d_2, ...l_k : d_k)$
$OFS(a)=(l_{k+1} : d_{k+1}, l_{k+2} : d_{k+2}, ...l_{k+r} : d_{k+r})$
$FIRE(a) : d_1, d_2, ...d_k \longmapsto Boolean;$
$PRE(a) : d_1, d_2, ...d_k \longmapsto Boolean;$
$POST(a) : d_1, d_2, ...d_k, d_{k+1}...d_{k+r} \longmapsto Boolean$
end a

According to the definition of the non-primitive actor in the last subsection, we have refinement $a \doteq (a_1, a_2, ...a_n)$. Ignoring the relationships between the actors at the right hand of the refinement, let

$$Abody = \{a_1, a_2, ...a_n\}$$

Then, for all $a_i \in Abody$, $i = 1, 2, ...n$, the representation of a_i can be

actor spec a_i **is**
$IFS(a_i) = (l_{i1} : d_{i1}, l_{i2} : d_{i2}, ...l_{im} : d_{im});$
$OFS(a_i) = (l_{im+1} : d_{im+1}, l_{im+2} : ...l_{im+s} : d_{im+s});$
$FIRE(a_i) : d_{i1}, d_{i2}, ...d_{im} \longmapsto Boolean;$
$PRE(a_i) : d_{i1}, d_{i2}, ...d_{im} \longmapsto Boolean;$
$POST(a_i) : d_{i1}, d_{i2}, ...d_{im}, d_{im+1}...d_{im+s} \longmapsto Boolean$
end a_i

Furthermore, according to the signature above, for any actor $x \in \{a, a_1, a_2, ..., a_n\}$, let

$$INPUT(x) ::= \{l_1, l_2, ...l_m\};$$
$$OUTPUT(x) ::= \{l_{m+1}, l_{m+2}, ...l_{m+s}\};$$
$$STATE(x) ::= \{x \| x \in (INPUT(x) \cup OUTPUT(x)) \wedge \text{ x is a state link}\}$$

Then, for any actor a, as previously specified, there are some rules to be followed by us for the actor representation, see figure 3.4, although the consistency can only be checked manually rather than automatically.

(1) $\forall\ l\ \bullet l \in INPUT(a) \Longrightarrow \exists a_i \in Abody, (l \in INPUT(a_i));$
(2) $\forall\ l\ \bullet l \in OUTPUT(a) \Longrightarrow \exists a_i \in Abody, (l \in OUTPUT(a_i));$
(3) $\forall\ l,\ a_i\ \bullet a_i \in Abody \wedge l \in OUTPUT(a_i) \wedge l \notin OUTPUT(a)$
$\wedge\ l \notin STATE(a_i) \Longrightarrow \exists a_j \in Abody\ \bullet\ l \in INPUT(a_j);$
(4) $FIRE(a_i)$, $PRE(a_i)$ *and* $POST(a_i)$ *must be consistent*
with $PRE(a)$ *and* $POST(a)$.

Figure 3.4. The rules for actor representation.

Rules 1 and 2 require that the role of an *external* link of an actor should be played by one of the links appearing in its refinement part, if any. An external link refers to a link appearing in the specification part of the actor. Rule 3 requires that an *internal link* (except state link) of an actor must be a connection between two actors of its refinement. The internal link of an actor refers to the link appearing in the refinement of the actor but not appearing in the specification part of the actor at the same time. Rule 4 requires the specification to be consistent with the predicates on the external links in the specification part of the actor and the predicates on the (external) links in the refinement part of the actor, which will be discussed in chapter 5.

Finally, we discuss the representation of primitive actors. The specification part of a primitive actor is the same to the specification part of a non-primitive actor. However, the refinement part of a primitive actor is a program text in some programming language.

3.2.4 Actor and EDFG

In the previous section the rules for representing actors have been addressed. The rules stipulate the connections between different actors in an actor body. In this section we discuss a method to represent the connections of different actors in an actor body.

The rules for representing a refinement require that an internal link connects two actors in the actor body. However, the connections between the actors inside an actor body are not yet reflected in the specification of an actor body as expressed previously. These connections can be represented with an additional feature, called *edfg*. The purpose of using the additional feature rather than expressing the connection inside the actor definitions (or specifications) is to keep the actor definitions independent of their application. An *edfg* is a part of Actor body, which specifies the connections between actors as expressed by the links of the actors in an actor body. The structure of an actor representation, containing an *edfg*, is (schematically) described as in figure 3.5.

The key word **type** in figure 3.5 is used to define link (data) types, which is similar to the definition of a data type in Ada, since link types are defined by means of data types. The key word **link** is used to declare **links** and their types. The expression

$$a_i(\textbf{input} :l_1, l_2, ...l_n; \textbf{output} :l_{n+1}, l_{n+2}...l_{n+m})$$

represents any instance of actor a_i. The key word **input** indicates the parameters of $IFS(a_i)$, and the key word **output** indicates the parameters of $OFS(a_i)$.

In an *edfg*, two actors are connected to each other if the same link appears in the **input** links of one actor and in the **output** links of another actor at the same time. A characteristic of using an *edfg* is that, for all *edfg*, there is an **EDFG** (Extended Data Flow Graph)[65] and vice versa. An EDFG, as represented in terms of a diagram[66], is a labeled graph which consists of actors (A), links (L) and arcs (E), where

$$E \subseteq (A \times L) \cup (L \times A)$$

An example of the correspondence between an edfg and an EDFG is given in figure 3.6.

```
actor spec a is
   ...
end a;
actor body a is
type d11 =...
type d12 =...
  ...
   actor spec a1 is ...
   actor spec a2 is ...
      ...
   actor spec ai is
   IFS(ai) = (l1 : d1, l2 : d2, ...ln : dn);
   OFS(ai) = (ln+1 : dn+1, ln+2 : dn+2, ...n + k : dn+k);
   FIRE(ai) : d1, d2, ...dn ⟼ Boolean;
   PRE(ai) : d1, d2, ...dn ⟼ Boolean;
   POST(ai) : d1, d2, ...dn, dn+1...dn+k ⟼ Boolean
   end ai
      ...
   actor spec an is ...
begin edfg
link l11, l12... : d11;
link l21, l22... : d12;
  ...
sink ln1, ln2...lx... : dn2;
sink ln+1,1, ln+1,2... : dn+1,2;
  ...
   a1(input :...; output :....);
   a2(input :...; output :....);
      ...
   ai(input :...lx...; output :....);
      ...
   aj(input :...; output :...lx....);
      ...
   ar(input :lr1, lr2, ...lrt; output :lrt+1, lrt+2...lrt+m);
      ...
   an(input :...; output...)
end edfg
end a
     di ∈ {d11, d12, ...}   i = 1, 2, ...n + k
     lri, lx ∈ {l11, l12...l21, l22...}   i = 1, 2, ...t + m
```

Figure 3.5. An actor.

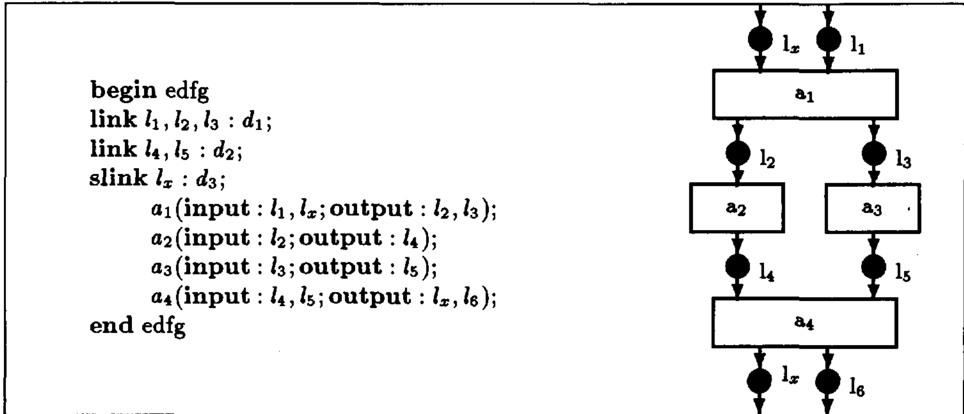


Figure 3.6. edfg and EDFG.

3.3 System architecture

In this section we discuss how a system architecture is built in terms of the concepts and the expressions of actors and EDFGs, and how system architectures can be reused to improve the process of system development and maintenance.

3.3.1 Creating a system architecture

A system architecture, as defined before, is the organizational structure of a system or a component. According to our approach, such an organizational structure can be represented in terms of different levels of EDFGs and their specifications, which can be obtained during the process of system design in the EDFG approach.

According to the EDFG approach, a highly abstract EDFG is created at the very beginning of the design process as described in 3.1. Such an EDFG will be refined into different levels of EDFGs, representing different levels of abstraction of a system. Following the design process, we can organize the different levels of EDFGs and their formal specifications into a hierarchical structure and maintain the hierarchical structure in terms of a semantic data base and additional tools, which will be discussed in 4.4 further.

In the hierarchical structure, each node represents an EDFG which consists of a set of actors and the relationships between them. Each actor of the EDFG has a formal specification. The refinement of each actor is presented at a lower level of the structure in terms of a lower level EDFG. An architecture of a system, for example, can be sketched as in figure 3.7.

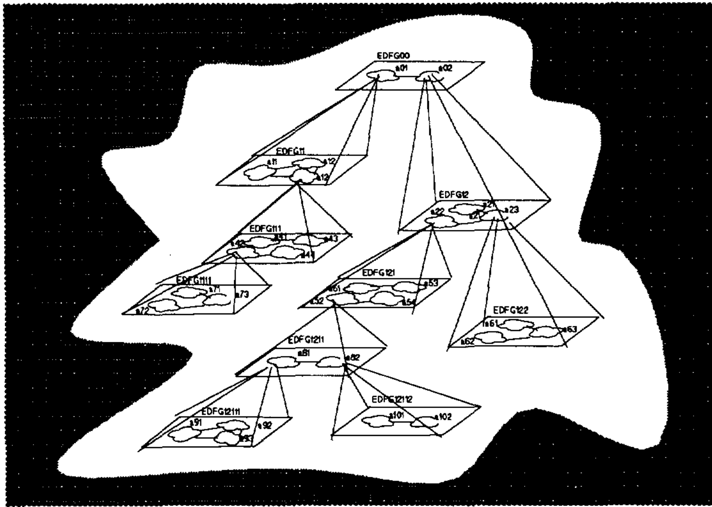


Figure 3.7. The system architecture.

We see that a system consists of two subsystems, which are expressed with two interrelated actors at the highest abstraction level. The two actors are refined recursively level by level. The actors which are not further refined are the primitive actors which can be implemented in terms of code components.

3.3.2 Reusing system architecture

Apart from the capability of binding small code components into a whole system, the reusability of an architecture also implies the capability of supporting the maintenance, evolution and reconstruction of the architecture, aiming at varieties of design target in an application domain.

Understanding complex systems

It seems that a human's capability of dealing with complex information is limited by the 'magic number seven', according to Miller[87][1]. However, complex information can always be understood in terms of abstraction. This is one way of representing the architecture of a system or a component in terms of different levels of abstraction as shown in figure 3.7. In order to overcome the difficulty of understanding complex systems, a designer may produce EDFGs containing no more than seven actors and organize them into the architecture. The process of understanding a system design may be realized through tracing the architecture. Tracing refers to finding all decomposed system designs and their dependencies, so that a system developer can easily understand, control and manage the design[52]. Tracing can be supported in terms of a tool which can be implemented, based on a data base in which the system architecture is maintained. An example of such a tool is described in chapter 7.

The flexibility for design modification

In principle, an abstraction, according to Shaw[107], is a specification of an object that emphasizes some properties of the object while suppressing others. When an actor is specified at a high-level in terms of an abstraction, it can be refined, based on lower-level design decisions. In this case, providing alternative lower-level design decisions result in alternative refinements. The alternative refinements may lead to alternative designs and implementations of the system.

From a reuse point of view, a higher-level specification can be shared by alternative refinements. Therefore, representing a system design in terms of different levels of abstraction as shown in figure 3.7 provides *flexibility* to modify the system design into an alternative one.

Enhancing design process

It is known that source code components may be used to improve the process of system implementation. However, source code components may not be very helpful to system design for a large and complex system[18][99][43]. We are

[1] According to Miller[87], the short-term memory of an average person has a limit of seven plus or minus two items.

going to describe how the architecture as defined in our approach can be used to improve the process of system design.

In contrast with the design process of the EDFG approach as described at the beginning of this chapter, the design process with reusing a system architecture can be described as follows. The process may be started from reviewing the top-level EDFG and its specification. A system developer needs to evaluate whether the new system also needs to be decomposed in the same way as shown in the EDFG at this level, and whether the new system has the same requirement to that of each existing subsystem, i.e. an actor of the EDFG. If the answer is yes, one may look into the refinement of each subsystem. At the refinement level, corresponding to each subsystem, one needs to evaluate whether the new subsystem can be built by reusing the existing design at this level, which is similar to the activities at the top-level. The process can be continued level by level until the reuser is satisfied with the design at a level and does not care about the detailed design or implementation, or until the reuser finds a refinement or a specification which is necessarily to be modified or replaced.

It seems quite different from reusing only small code components that reusing architectures can obtain additional savings from the knowledge about how to analyze and design a similar system, from the awareness about binding small components into a system and from the easiness of understanding and maintaining a system.

3.4 Templates of system design

In the previous sections we see how system architectures provide vertical hierarchies for structuring system design. In this section we deal with horizontal hierarchies of system design. While vertical hierarchies in our method refer to different levels of abstraction, horizontal hierarchies refer to the template of system designs and some relative features supporting reusability, such as *import* and *inheritance.*

A template is a pattern of a set of similar actors or EDFGs at the same abstraction level. Import refers to using the textual resources defined in other components. Inheritance can be realized by using the textual resources previously defined in the hierarchical structure representing system architecture.

3.4.1 Template

The term *template* is defined in different ways in literature. It may be defined as "any fixed theme plus the means to accommodate unforeseen variations on the theme"[10], or as "an algorithm specification that is free from the commitments to specific types and representation of data"[122]. A template can be formally defined as a functor which is a mapping from objects and morphisms in one category to objects and morphisms in another, preserving morphism source, morphism target, and the actions of identity and composition[80]. In our approach, a template is defined as a mapping from components to components. The components in this chapter are restricted to the design representations at different levels of design decomposition. The design representations might be given in the forms of Extended Data Flow Graphs(EDFGs) or the specification of actors[66].

The actor type. In order to represent template of EDFGs and actors, let us first represent the pattern of actors, i.e. a specification of a set of alternative actors. For example, assume a system includes an actor which represents a device. In order to apply the same design of the system to different application environments, the actor can be alternatively defined with different input links such as *l:char / l:line / l:file* . In this case a generic specification is required. Such a specification is called an *actor type*. An actor type is an abstraction of a set of alternative actor specifications.

Since the properties of an actor are determined by the attributes of the 5-tuple, an actor type can be defined with a generic specification accordingly. The structure of the generic specification is described by

Atype AT **is**
$IFS(AT) = (l_1 : G_1, l_2 : G_2, ...l_n : G_n);$
$OFS(AT) = (l_{n+1} : G_{n+1}, l_{n+2} : G_{n+2}, ...l_{n+k} : G_{n+k});$
$FIRE(AT) : G_1, G_2, ...G_n \longmapsto \textit{Boolean};$
$PRE(AT) : G_1, G_2, ...G_n \longmapsto \textit{Boolean};$
$POST(AT) : G_1, G_2, ...G_n, G_{n+1}...G_{n+k} \longmapsto \textit{Boolean}$
end AT

The $G_1, G_2, ...G_{n+k}$ are link types or **generic link types** which are defined as follows.

definition 2 *A generic link type is a* **universal** *link type or a* **union** *link*

type. A link type is said to be *universal* if and only if each link of the type is capable of accepting the tokens of any data types. A *union* link type is a link type which allows every link of the type to accept the tokens of different data types for different application. A generic link type can be denoted by

<*Generic-link-type*> =
U-type |
'[' link-type$_1$*, link-type*$_2$*,.... link-type*$_n$ *']'*

where U-type stands for universal link type.

In terms of generic link types, IFS(AT) of the actor type AT can be rewritten as in figure 3.8. Similarly, OFS(AT) can also be rewritten. The representations of the rewritten IFS(AT) and OFS(AT) are useful to understand the relationship between an actor type and a particular actor specification of the type.

The most direct application of an actor type is the definition of the formal parameter of a template. The basic operations on an actor type include: (1) creating actors which *bind to* a given actor type, and (2) verifying whether a given actor satisfies a given formal actor type. We will use the predicate $Satisfies(a, AT)$ later on to denote "actor a satisfies actor type AT".

$$\text{IFS(AT)}=(l_1 : G_1, l_2 : G_2, ...l_n : G_n) =$$

$$\{(l_1 : d_1, l_2 : d_2, ...l_n : d_n) \mid \begin{cases} d_i = G_i & \text{if } G_i \text{ is a link type,} \\ d_i = g_j & \text{if } G_i = [g_1, g_2, ...g_m], \\ & \quad j \in \{1, 2, ...m\}, g_j \text{ is a link type } \}; \\ d_i = \text{ any link type,} & \text{if } G_i \text{ is a U_type.} \end{cases}$$

Figure 3.8. A rewritten clause of a generic type.

Template representation

The specification of a template can be represented in terms of a parameterized actor and its refinement. For example, the signature of a template, *Templ*, with actor parameters A and B of type AT is shown in figure 3.9, in which

```
actor  Templ (A, B: AT)
         IFS(Templ)
         OFS(Templ)
         FIRE(Templ)
         PRE(Templ)
         POST(Templ)
end Templ
actor body Templ is
         actor a1 is
                  IFS(a1)
                  OFS(a1)
                  FIRE(a1)
                  PRE(a1)
                  POST(a1)
         end a1
                  ...
         actor ai is A
                  ...
         actor aj is B
                  ...
         actor am is
                  IFS(am)
                  FS(am)
                  FIRE(am)
                  PRE(am)
                  POST(am)
         end am
begin edfg
         ...
end  edfg
end Templ
```

Figure 3.9. A template.

(A, B: AT) is a parameter list in which A and B are the formal parameters of the actors *Templ*, and AT is the actor type of the parameters. In the actor body, actor a_i and a_j are allowed to be any actor satisfying the actor type AT respectively.

Obviously, the representation of a template is a parameterized actor which has a pattern of actor bodies. The EDFG derived from the pattern of an actor body is a parameterized EDFG, which can easily be seen from the represen-

tation of a template and from the relationships between actors and EDFGs.

Instantiation

From the definition above we see that a template is a pattern of an actor. Such a pattern can be used to generate particular actor specifications which are known as instances of the pattern. The process of generating an instance of the pattern is known as *instantiation*, which is concerned with selecting alternative sub-actors to fulfill the pattern. An actor instantiation can be declared with an expression, e.g.

$$\textit{new-actor} = \textit{Templ(actor1, actor2)}$$

where *Satisfies(actor1, AT)* and *Satisfies(actor2, AT)* must be true. The expression above defines a new actor, *new-actor*.

3.4.2 Import and inheritance

In order to increase the possibility of an application reuse, mechanisms are required to allow the composition of a new component with existing resources of an application. Two of the fundamental operations for the composition are *import* and *inheritance*, which are discussed in this subsection.

Import

An import allows the textual resources of imported actors to be logically *owned* by an importing actor, therefore, reducing the effort of constructing the importing actor. The textual resources of an actor refer to the textual representation of the actor or its parts, including link types, sub-actors and edfg. A resource is said to be owned by an actor if and only if the resource is available in the actor as if it is defined in the actor.

An import is described by an import clause appearing in the specification of an importing actor in the form of

$$\textbf{import}\ a_1\ [(parameter_list)], a_2[(parameter_list)]...a_n[(parameter_list)]$$

The *parameter-lists* are optional, which allow the import of parameterized actors. There are several rules for import. Assume actor a is imported into actor b.

1. *If a is a primitive actor and b is a non-primitive one, a is available to be used to construct the edfg of b.*

2. *If both a and b are primitive, the import rule might depend on concrete details of the programming language, which is out of the discussion of the chapter.*

3. *If both a and b are non-primitive, all resources of a are available in b, i.e.,*

   (a) *every actor owned by the actor body of a is owned by the actor body of b;*

   (b) *all type definition owned by a are owned by the actor body of b;*

   (c) *the edfg owned by a is owned by b, but it can only be reused as a whole to construct the edfg of b.*

4. *If a is a template, it can be applied to define one or more actors owned by b. However, the actual parameter of the template must be the actors which are owned by b.*

In order to give an intuitive impression about the notion import, an example is provided. The key word **import** in the example indicates that template a_{k1}, and actors a_{k2} and a_{k3} are imported into b. Actor a_j is defined by instantiating template a_{k1} with actual parameter a_1 and a_{k3}. The *edfg* of actor a_{k2} is applied to form the *edfg* of b.

```
actor spec b (parameter-list) is
    import a_k1(parameter_list_1), a_k2, a_k3;
        ...
end b
actor body b is
    type ...
        actor spec a_1 is ...
        actor spec a_2 is ...
            ...
        actor spec a_j is a_k1(a_1, a_k3);
            ...
        actor spec a_n is ...
    begin edfg
        link ...
            a_1(...);
            a_2(...);
              ...
            a_k2(...);
              ...
            a_n(...);
    end edfg
end b;
```

Figure 3.10. Template with import.

Inheritance

Inheritance is a technique for sharing resources following a hierarchical structure, which provides means to define system components as extensions of previously defined ones. There are two kinds of inheritance in the representation of actors.

First, the refinement of actors must preserve the relationships between them. Applying such a principle to the architecture of a system or a component, we see that the relationships between actors in an *edfg* defined earlier in the hierarchical structure are inherited by the refinements of the *edfg* level by level.

Secondly, although it is legal to define (external) link types inside an actor body and to define the same actor in different actor bodies, this is often

redundant because such definitions may already be defined in its parent (node) or ancestors in the hierarchical structure. The data and the actor definition appearing in an actor body can be inherited by its descendants.

3.5 Summary

In this chapter a method was presented for an explicit representation of system design. Such a representation includes both a vertical and a horizontal hierarchy, the former representing system design in terms of different levels of abstraction, the latter representing the parameterization of design representations. Each of them tells us how a complex application system or system component can be composed from lower level entities.

By an explicit representation of system designs, a complex system or a large-grain component can easily be understood and manipulated, and can easily be maintained and reconstructed to meet varieties of requirements.

Chapter 4

The reuse of large-scale components

The change from a program-centered to a data centered view of programming is comparable to the shift from the earth-centered to the sun-centered view of the solar system brought about by the Copernican revolution.

— Charles Bachman, Turing Lecture, 1973.

In the previous chapter a method was described for the reuse of system architectures and design templates, so that reuse-in-the-large is realized. However, such a method is limited by a particular approach for application development, i.e. the EDFG approach. The reuse is limited to reusing the representation of a single system rather than that of an application domain, and there is no specification of the resources to be reused when reuse is needed for the maintenance, evolution or reconstruction of the system. In this chapter we discuss the reuse of large-scale components. Our claim is that the reuse of large-scale components is capable of solving the problems above and capable of improving the process of analysis, design, implementation and maintenance.

The remaining part of this chapter is organized as follows. In section 1 we identify the notations for modelling large-scale components. In section 2 we introduce the pragmatic model of large-scale components. In section 3 we present the operating model of large-scale components. In section 4 we describe the management and specification of large-scale components respectively. In section 5 we discuss the capabilities of reusing large-scale compo-

nents. In section 6 we address an information system in the form of a support environment for reusing large-scale components. Finally, in section 7 we draw a conclusion.

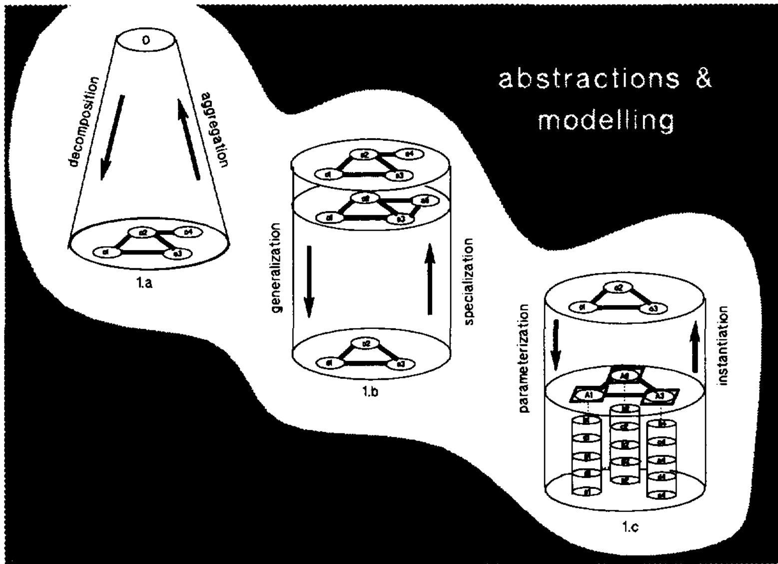


Figure 4.1. The abstractions for component modelling.

4.1 The abstractions for component modelling

System components are a kind of data or can be described in terms of data, indicating the possibility of applying the notations for data modelling to the practice of component modelling. For modelling a large-scale component, we borrow notations from both semantic data modelling [13] and application development including *aggregation, decomposition, generalization, specialization, parameterization* and *instantiation.* These notations have formed the basis to describe our approach for the reuse of large-scale components.

Aggregation and decomposition are two complementary activities which relate the similar abstractions from opposite directions. *Aggregation* abstracts the composition of several objects to a higher-level aggregate object[13]; *decomposition* represents a high level object in terms of a set of interrelated

lower-level objects. The abstractions of aggregation and decomposition are described in figure 4.1.a.

Generalization and specialization are another pair of complementary activities which relate abstractions from different directions. *Generalization* abstracts a number of common characteristics of a collection of objects to a generalized object[13]; *specialization* composes some special characteristics with the common characteristics to a specialized object. The abstractions of generalization and specialization are described in figure 4.1.b.

Parameterization and instantiation are the third pair of complementary activities which relate abstractions in different forms. Parameterization abstracts an object to a pattern, which is normally done by replacing some elements (sub-objects) of the object with their abstract forms. The pattern can be reused to form various similar objects. In contrast with parameterization, instantiation fills the pattern with special elements (sub-objects), so as to produce a specific object. The abstractions of parameterization and instantiation are described in figure 4.1.c.

Generalization and parameterization. Parameterization is considered to be a kind of generalization in literature [106]. We prefer a different categorization since a generalization captures the basic features of a component, whereas parameterization provides a pattern for component composition. The distinction between generalization and parameterization is very important in this dissertation. Sometimes we need to generalize a component first and then parameterize the generalized component, which will be described more precisely in section 4.3.5.

Instantiation and specialization. Instantiation is an activity to generate an instance of a kind of component with respect to a parameterized component. But an instance can also be a generalized component which needs sometimes to be specialized further. An example for such a specification is provided as follows.

```
(1) editor(W: Window; l: Link; k: Keyboard).
(2) constructor( d: Drawing; e: Editor).
(3) basic-editor = editor(multi-window, multi-link, std-keyboard).
(4) graphic-editor =constructor(drawing, basic-editor)
```

Assume — in the example above — that the *editor* in (1) and the *constructor* in (2) are the signatures (headings) of two parameterized components. The *basic-editor* in (3) is an instance from the instantiation of the *editor* in (1). Such an instance is a generalized component in contrast with the *graphic-editor* in (4), in other words, the *graphic-editor* in (4) is a specialization of the *basic-editor* in (3).

4.2 The pragmatic model of large-scale components

A large-scale component is the combination of both a large-grain code component and its design information, which is represented integrately in terms of different levels of abstraction. A large-scale component (LSC) is defined as a 4-tuple,

$$LSC ::= (\textit{design-framework}, \textit{design-instance} \\ \textit{domain-resources}, \textit{refinement})$$

A *design framework* describes how to integrate lower-level entities (objects and relationships) into alternative design instances. It is made up from a set of *abstract* objects and the relationships between them represented in terms of a module interconnection language[99], e.g. EDFG (Semantically Extended Data Flow Graph)[74].

A *design instance* is an instance of the design framework or a specialization of the instance of a design framework. A design instance consists of a *specification* and an *architecture*. The specification specifies the semantics of the design as well as the syntax and semantics of its interface in terms of a specification language. The architecture describes the structure of the design in terms of (relatively) particular objects and their composition in terms of a model interconnection language.

A set of *domain resources* consists of a *domain model* and an associated set of *domain entities*[96] which can be used for the instantiation of the design

framework. The domain model consists of a set of attributes defining the requirements (or constraints) for the objects appearing in the design framework. The definition of the requirements (or constraints) may be provided in terms of a specification language. The domain entities provide instances (or values) of the attributes already known to exist.

A *refinement* refers to the decomposition or the implementation of the design instance. A refinement consists of a set of (lower level) large-scale components or implementations. Each of the (lower level) large-scale components or implementations is an elaboration of an object appearing in the design instance which is viewed as a set of objects and the relationships between them. Such an elaboration inherits the semantics of this object and relationships between this object and others. The refinement may be recursively continued until all objects in the design instance are suitable to be implemented in terms of small components, such as objects in an object-oriented programming language or packages in Ada.

The pragmatic model of a large-scale component is a hierarchical structure (H-structure), as in figure 4.2. Each node of an H-structure contains three pieces of information: a design framework, a design instance and a set of domain resources. The relationship between the three pieces of information is that the design framework is a *mapping* from a set of domain resources to a range of design instances. An H-structure is defined recursively, in which each lower level node is the detailed design of an object appearing in a higher level design instance (not design framework and domain resources). Such an organization not only allows a large-scale component to represent a target component but also makes the different levels of the design frameworks and design resources independent of each other.

Additionally, the nodes in an H-structure, corresponding to small code components, are called *terminal nodes*, otherwise, *non-terminal nodes*. A terminal node of an H-structure need not to be refined into lower-level large-scale components, whereas a non-terminal node does.

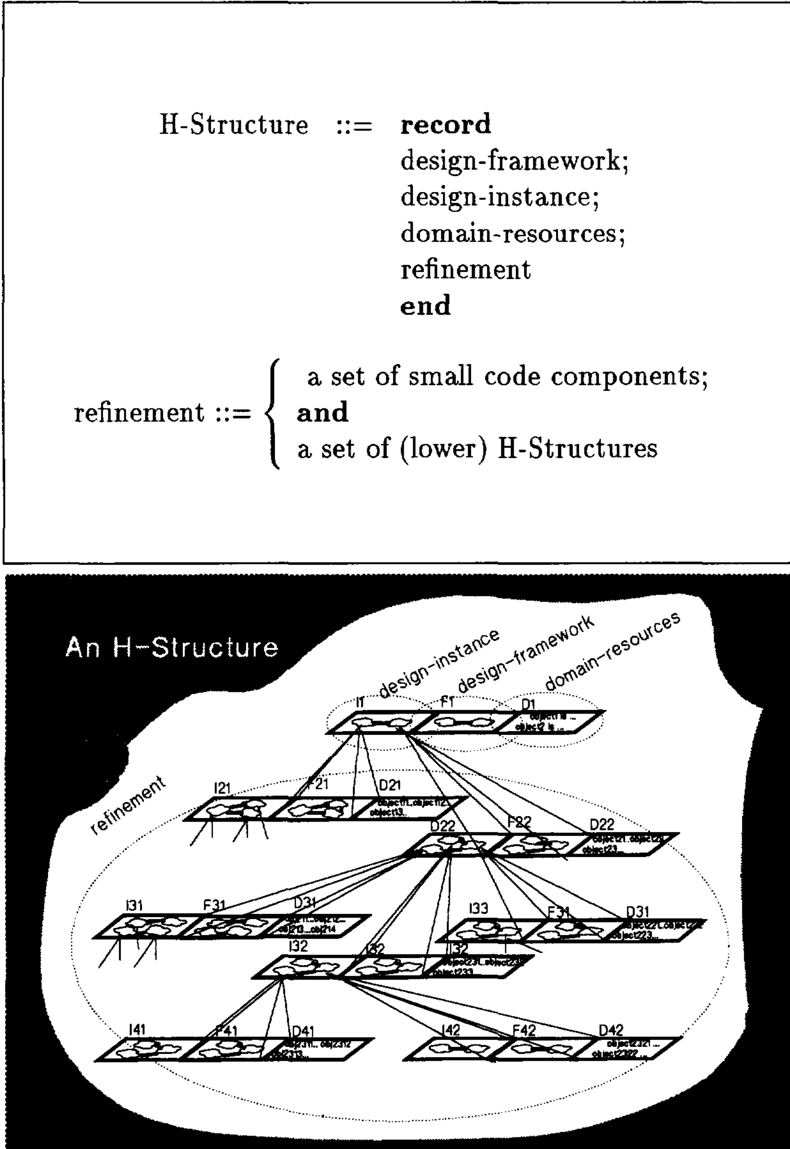


Figure 4.2. The pragmatic model of a large-scale component.

Obviously, an H-structure provides a clear organizational structure of a large-scale component. The interconnections between the constituent parts of the components are especially useful for designing alternative large-grain

components (subsystems or even systems) in an application domain. For example, if a similar design is needed for a new application, it can be obtained by modifying the existing design instance. First, the existing design (concerning all design instances of a large-scale component) can be understood according to different levels of abstraction. Secondly, the instances to be modified or replaced can be located by browsing the H-structure. Thirdly, once an alternative design instance is needed, there are domain resources and a design framework. The former provides lower level entities the design instance may be composed from; the latter suggests how the design instance can be composed with the entities.

4.3 The operating model of large-scale components

In order to support reuse-in-the-large, different views are needed to model a large-scale component. Whereas the H-structure of a large-scale component captures the organizational structure of the component, an operating model of a large-scale component captures the logic relationships between different parts of the component. From an operating model we see how different parts of a large-scale component depend on each other. The operating model of a large-scale component can be described in terms of the activities applied to the creation and manipulation of the constituent parts of a large-scale component, as shown in figure 4.3.

Figure 4.3 consists of five forms of component representation: an object (top), a design instance (left), a design framework (right), a set of domain resources (bottom) and an *interim instance* (center). The object at the top of the figure is assumed to be an object of a design instance at an abstract level. The design instance, design framework and the domain resources form one-level refinement of the object. The interim instance, which may also be called interim design instance, is a transient component for illustrating the relationships between the other four forms of component representation. The relationships between the different forms of representation are described in terms of labeled arrows. The arrows show the transformation or dependency between the different forms of representation, and the labels describe the actions concerned with the transformation or dependency.

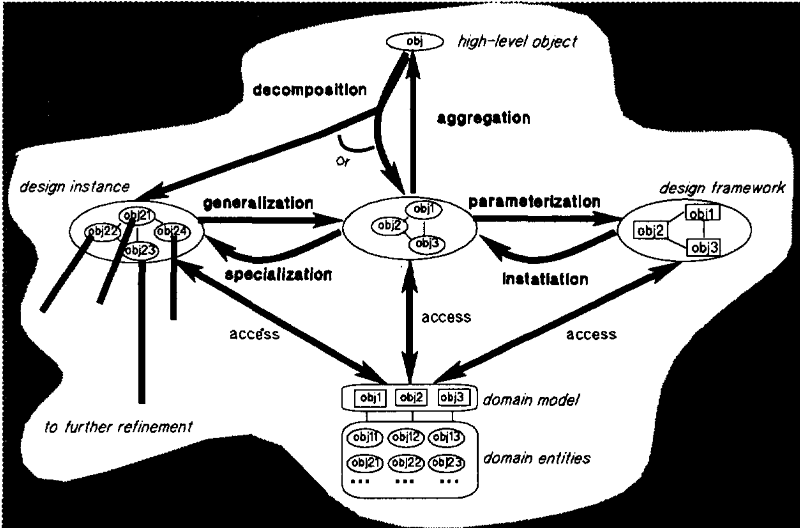


Figure 4.3. The operating model of a large-scale component.

4.3.1 The relationship between an object and its refinement

As shown in figure 4.3, the relationship between an object and its refinement can be given in terms of decomposition and aggregation.

In a large-scale component a high-level object represents an abstraction. Such a high-level abstraction can be decomposed into several interrelated lower-level abstractions. The elements of such a decomposition can be further decomposed into more lower-level abstractions until the resulting abstractions can easily be implemented in terms of small code components.

The opposite result can be obtained by aggregation. In terms of aggregation we identify several interrelated objects, then raise an abstraction representing a higher-level object.

Although both decomposition and aggregation result in similar structures regarding the relationship between an object and its refinement, decomposition is useful to simplify the complexity of components to be designed, whereas aggregation is useful to utilize existing resources.

4.3.2 A design framework: a parameterized design

A design framework is the encoding of the information about the composition of a system design from its lower-level entities. A design framework is used as a basis for the construction of design instances according to the requirements on the instances to be built. A design framework itself should be both generalized and parameterized. The representation of the design framework should be generalized to capture the common characteristics of the different instances of the design. It should be parameterized so that various instances can be generated by filling a pattern with given objects.

A design framework can be constructed by parameterizing a generalized design, either originated from the decomposition of an object or from the generalization of a design instance.

4.3.3 Domain resources: objects for instantiation

A domain model represents the requirements from the application area, and is the necessary part for the parameterization of a generalized design. The parameterization of a design implies formulating constraints on the objects appearing in a given design, in order to allow the actual substitution for these objects in terms of other objects satisfying the constraints. The specification of these constraints is the domain model.

A domain model is specified by giving the specifications for the constraints on the objects appearing in the design framework. These specifications are useful to check whether a group of objects are feasible to form a legal design by the instantiation of the design framework. The specifications are also useful to identify the candidate objects for alternative instantiations of the design framework. Such candidate objects are known as domain entities.

4.3.4 Design instance: a specialized design

A design instance is a delegate of all the possible designs at an abstract level covered by the respected higher-level object. Such a delegate is not intended to present the common behaviors of all possible designs but, instead, to catch the particular characteristics of a design for a target system or component. Such a delegate can be either generated directly by the decomposition of a higher-level object or by a specialization of a design resulting from the instan-

tiation of a design framework.

```
scheme LARGE-SCALE-COMPONENT(O:OBJECT, P:PRIMITIVE,
                    F:FRAME, D:DOMAIN, I:INSTANCE)=
class
  type
    Object = O.Object
    Primitive = P.Primitive
    Frame = F.Frame
    Domain = D.Element
    Instance = I.Instance
  value
    parameterize: Instance ⟶ Frame × Domain
    instantiate: Domain × Frame ⟶ Instance
    generalize: Instance ⟶ Instance
    specialize: Instance ⟶ Instance
    decompose: Object ⟶ Instance
    aggregate: Instance ⟶ Object
  axiom forall i,inst:Instance, f:Frame,d:Domain, o:Object •
    (1) parameterize(i) as df
            post ∃ d:Domain, f:Frame • df = (d, f)
            pre ∃ o:Object • aggregate(i)= o ∧ generalize(i)=i
    (2) instantiate(parameterize(i)) ≡ inst
            post ∃ f:Frame, d1,d2: Domain •
                    parameterize(inst)=(f,d1) ∧ parameterize(i)=(f,d2)
            pre ∃ o:Object • aggregate(i)= o ∧ generalize(i)=i
    (3) specialize(generalize(i)) as inst
            post generalize(inst)=generalize(i)
    (4) generalize(specialize(i)) ≡ i
    (5) decompose(aggregate(i)) as inst
            post aggregate(inst)= aggregate(i)
    (6) aggregate(decompose(o)) ≡ o
end
```

Figure 4.4. The specification of the operating model.

4.3.5 The formalization

With the notations addressed in previous sections and following the previous discussion in this section, the operating model can be formalized in terms of the specification language RSL[47], see also figure 4.4. Such a formalization aims at providing an exact description of the relationships between the different constituent parts of a large-scale component and building a basis for developing a set of tools supporting reuse-in-the-large, although the latter will not be discussed in this chapter.

In figure 4.4, we assume that the types *Object, Primitive,Frame, Domain* and *Instance* have been defined somewhere else. The detailed specification of them is not important here, but will be discussed in chapter 5.

In this specification the operating model of a large-scale component is specified in terms of transformation between its constituent parts: objects, design instance (instance), design framework (frame) and domain resources (domain). The transformation is built in terms of several values [47] including *parameterize, instantiate, generalize, specialize, decompose, and aggregate.* The six axioms in figure 4.4 describe the constraints on the transformation.

Axiom (1) implies that an interim instance can be parameterized into a design framework and domain resources. An interim instance is a design instance which satisfies the precondition of the axiom.

Axiom (2) implies that, if an interim instance is parameterized into a design framework and domain resources, alternative interim instances with respect to the same design framework can be generated by means of instantiation. Note that the domain resources might be changed by putting (or removing) domain entities in them dynamically.

Axiom (3) implies that, if a design instance is generalized, alternative design instances with respect to the generalized design instance can be generated by means of specialization.

Axiom (4) implies that, if a design instance is specialized, its original representation can be recovered from the specialized design instance by means of generalization.

Axiom (5) implies that, if a design instance is aggregated into a high-level object, alternative design instances with respect to the high-level object can be generated by means of decomposition.

Axiom (6) implies that, if a high-level object is decomposed into a design instance, the high-level object can be recovered from the design instance by

means of aggregation.

According to the axioms in the specification and considering the operating model as shown in figure 4.3, we can explicitly describe how the constituent parts of a large-scale component are generated, i.e.

high-level-object	$\Leftarrow$	*aggregate(interim-instance)*
	$\Leftarrow$	*aggregate(design-instance)*
interim-instance	$\Leftarrow$	*decompose(high-level-object)*
	$\Leftarrow$	*generalize(design-instance)*
	$\Leftarrow$	*instantiate(domain-resources, design-framework)*
design-framework	$\Leftarrow$	*parameterize(interim-instance)*
domain-model	$\Leftarrow$	*parameterize(interim-instance)*
design-instance	$\Leftarrow$	*interim-instance*
	$\Leftarrow$	*specialize(interim-instance)*
	$\Leftarrow$	*decompose(high-level-object)*

The table tells us that the different constituent parts of a large-scale component can be constructed from each other. This implies flexibility and reusability concerning the maintenance, evolution and reconstruction of a large-scale component to meet variety of requirements. For example, in order to change a design instance in a node of an H-structure, one may (1) directly produce a new design instance, (2) modify the original design instance, (3) generate an interim instance and use-as-it-is, or (4) generate and specialize an interim instance. Moreover, in order to get the interim instance, one may (5) generalize the original design instance, (6) decompose the high-level object, or (7) instantiate the design framework with the domain resources.

Practically, we think that the operating model of large-scale components provides a guideline for building and composing the constituent parts of the components.

4.4 Management of large-scale components

In this section we discuss how large-scale components can be represented in terms of a semantic database and how the components can be reused.

4.4.1 Retrieval problem

One of the key problems of component management in application reuse is retrieval. Quite often it is easier for a potential user to build a component from scratch than to find the component in a library and understand the constraints on its use [93]. Although the facet classification[102] by Prieto-Diaz is a good method for component retrieval, especially the problem of component retrieval, the problem of semantic matching between user's requirement description (e.g. semantic specification) and the specification of the components are far from being solved.

The retrieval problem can, however, in many cases be settled in particular application domains by using domain analysis e.g. Draco approach[93]. While most methods for component retrieval are centered on classifying components according to their functionality and properties, we organize the components in a library according to a process of system development which can be partially traced according to the H-structure of a large-scale component. As a result, both system components and the context for the application of the components are maintained. Obviously, the context information has been clearly described in the pragmatic model of large-scale components. The remaining problem is how to represent the organizational structure in a component management system, so that a large-scale component can be manipulated according to the operating model of the component.

4.4.2 Semantic data base and semantic data modelling.

Semantic databases allow dealing with semantics in terms of *conceptual models*[13]. A conceptual model, according to [13], is a predefined partial image of the real world. Conceptual models can be represented by *types* and their *assertions*. In this context a type is an aggregation of distinct attributes, including an object identity for naming the object of this type. Assertions are predicates, describing the properties of the attributes and the relationships between them. They can be used to express the relationships between a conceptual model and the real world. Any object satisfying the definition is known as an *instance* of the conceptual model.

Semantic data modelling refers to the use of semantic data models (conceptual models) for the description of the real world. For semantic data modelling, a useful theory is type algebra[13] which consists of semantic models (types) and a set of operations including generalization, specialization and

aggregation.

There are several principles for semantic data modelling, including *convertibility, relatability* and *object- relativity* [13]. Convertibility implies a one-to-one relationship between assertion subject and predicates. It is useful to avoid ambiguity of the concepts regarding the objects appearing in large-scale components. Relatability indicates that an attribute is related to one and only one type with the same name, and each type can play a role of attributes in various other types. If relatability is followed, a hierarchical structure can be built without ambiguity, which is necessary for representing the organizational structure of a large-scale component. Object-relativity indicates that the same abstract object can have different interpretations including type, instance, generalization, aggregation, specialization, and attribute. The different interpretations correspond to the operating model of large-scale components. The principles discussed above are useful for the representation of large-scale components, since system components are really a kind of data and can be represented in terms of the concepts from semantic data modelling.

4.4.3 The representation of large-scale components

The representation of a large-scale component describes the constituent parts of a large-scale component and their interconnections. Such a representation can be based on the notations provided in semantic data modelling.

The domain resources As discussed in 4.2, each set of domain resources in a large-scale component consists of a domain model and an associated set of domain entities. The domain model consists of a set of attributes defining the abstract objects appearing in the design framework. The domain entities provide the instances (or values) of the attribute. Assume that a design framework represents an editor and that the abstract objects appearing in the design framework are *window, keyboard* and *link.* The domain model of the editor can be defined in terms of a conceptual model as follows:

```
type editor = window, keyboard, link
assert editor =
        the specification of window
        the specification of keyboard
        the specification of link
```

Assume that the specifications in the conceptual model are provided in

terms of a semantic specification language and that the general requirements for each attribute are described semantically.

The domain model above can be used to identify domain entities. Objects which are feasible to an attribute specification are the entities (values) of the attribute. The entities from each attribute form a domain instance of the model. For example, a domain instance can be

graphic-editor = multi-window, graph-keyboard, multi-link.

In this case the domain resources of the editor correspond to a relation as follows:

editor	window	keyboard	link
...	...	...	...
graph-editor	multi-window	graph-keyboard	multi-link
...	...	...	...

The design framework. A design framework describes a set of *abstract* (or non-abstract) objects and the relationships between the objects. Since the abstract objects are already defined inside the domain model, the design framework is only necessary to specify the relationships between the objects. In this sense, a design framework can be viewed as additional semantic or syntactic constraints on a domain model, which, therefore, can be part of a domain model from a managerial point of view. A frame for representing both the domain model and the design framework is provided by an example as follows:

```
type  editor = window, keyboard, link
assert editor =
     domain
        the specification of window,
        the specification of keyboard,
        the specification of link.
    frame
        the relationships between window,
                keyboard and link.
end
```

The design instance. According to the operating model of a large-scale component, a design instance is a specialization of an instance of a design framework. The instance can be obtained by instantiating the design framework with the domain entities of domain resources. In this sense, a design instance is specific. However, a design instance might be generic as well since it can be a higher-level abstraction of a particular target design, i.e. the objects appearing in a design instance can be abstract objects for further decomposition. Therefore, a design instance is a type rather than only an instance. Fortunately, semantic data modelling allows an instance of one type to be a type.

For example, in the representation of design resources we see that a graph-editor is an instance of the type editor and all the sub-objects appearing in the instance may not necessarily be primitive concepts. We may, therefore, further provide the semantics of such an instance by a type definition

type graph-editor = multi-window, graph-keyboard, multi-link

As a graph-editor may contain more attributes than the basic attributes of an editor, such as *draw*, the definition above can be specialized by adding a new attribute, i.e.

type graph-editor = multi-window, graph-keyboard, multi-link, draw

The assertion of the graph-editor is as follows,

```
assert graph-editor =
   object
      the specification of a multi-window,
      the specification of a graph-keyboard,
      the specification of a multi-link,
      the specification of draw.
   instance
      the architectural specification of a graph-editor.
end
```

The specification of the graph-editor above is the design instance of the editor.

Refinement. Apart from the domain resources, design framework and design instance as represented above, we also need to describe the refinement in order to represent a large-scale component. Such a refinement allows us to

look into each object appearing in the design instance and to represent the domain resources, design framework and design instance of this object. This can be done by representing lower-level large-scale components in terms of the type notation recursively. From a managerial point of view, the structure of the refinement can be described as in figure 4.5. Figure 4.5 represents a large-scale component OBJX. The domain model of OBJX consists of the attributes OBJi, OBJj and OBJk. The constraints of OBJX contain the semantic specification of the domain model and the specification of the design framework. OBJA and OBJB are domain instances. The OBJB consists of objects objB1, objB2 and objB3, which is further specialized by adding objB4, leading to a design instance objBi. The semantics of the design instance are provided by its constraints, including an architectural specification. The design instance objBi is a type. Its attributes can be refined into lower-level large-scale components such as OBJY and OBJZ.

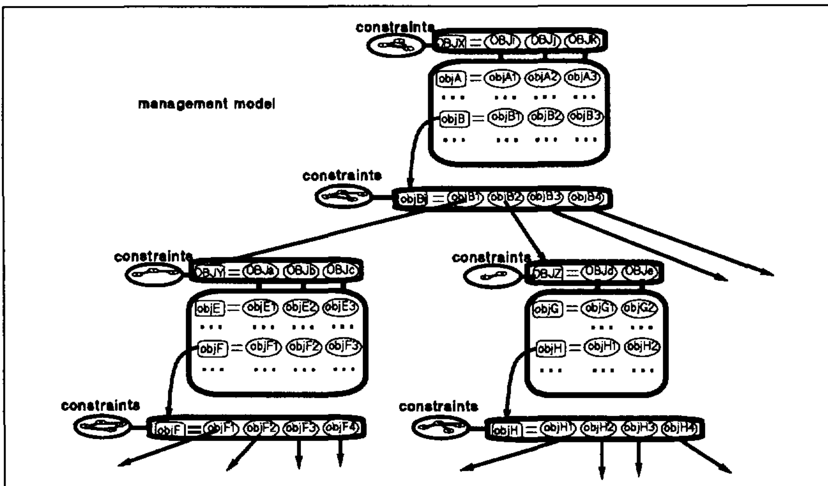


Figure 4.5. The refinement structure of a large-scale component.

4.4.4 The component base

The component base supporting the reuse of large-scale components can be a semantic data base as mentioned before. The semantic data base is not only capable of supporting the representation of large-scale components, leading to conceptual models, but also provides support to guarantee the semantic

integrity of the model in terms of semantic constraints (specification). The semantic constraints include inherent, static and dynamic constraints. The inherent constraints guarantee relatability and convertibility, the static

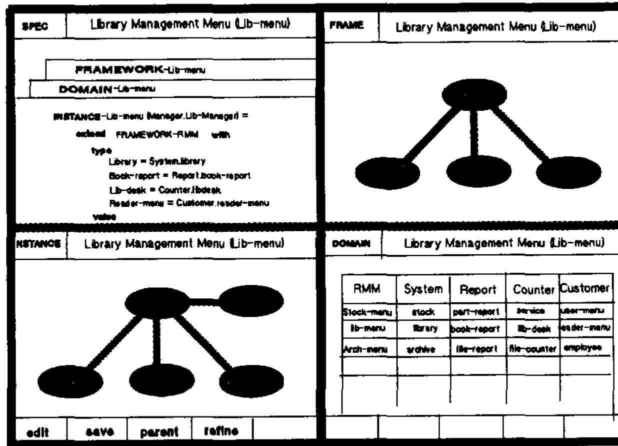


Figure 4.6.a. The user interface of a large-scale component.

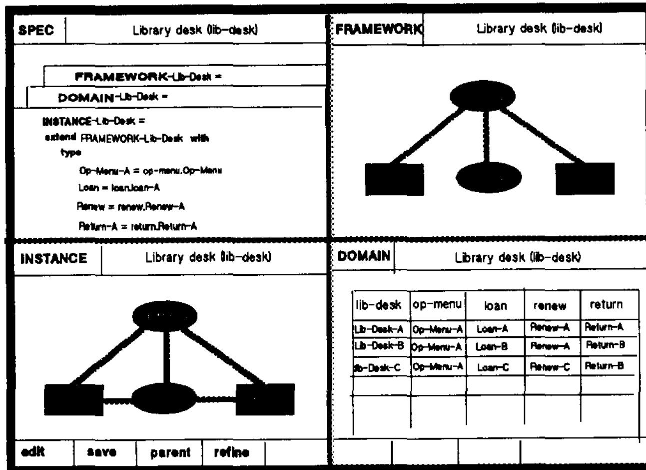


Figure 4.6.b. The user interface of a large-scale component.

constraints guarantee some statement to be true on a defined structure of a conceptual model, and the dynamic constraints guarantee the correctness of the operations including insert and update. The three kinds of constraints are useful to guarantee the correct representation and reuse of large-scale components.

Since large-scale components are represented in terms of conceptual models, the semantic operations can be applied to create, retrieve, modify and reconstruct the components. Moreover, additional tools are needed to support system developers and reusers to represent and reuse large-scale components.

One of the tools to be mentioned is a graphical structure as an interface between a large-grain component and reusers, which are prototyped as shown in figure 4.6.a and figure 4.6.b, the former presenting a node of a large-scale component at top-level, the latter representing a refinement of an object, appearing in the design instance of its parent node.

The figure 4.6.a and 4.6.b show a user view of a large-scale component. In Figure 4.6.a we see intuitively that a *Library Management Model*, a large-scale component, is displayed by four windows: specification (*spec*, left-top), design instance (*instance*, left-bottom), design framework (*frame*, right-top), domain resources (*domain*, right-bottom).

In window *frame* a design framework is displayed, which consists of four interrelated abstract objects: *system, counter, report* and *customer*. The textual specification of the design framework can be found in window *spec*. The design framework may be specified in terms of a module interconnection language, e.g. EDFG.

In window *domain* a domain model *RMM* (Resource Management Menu) is displayed, which consists of the abstract objects appearing in the design framework. The functional specification can be found in window *spec*. The domain entities are listed in a table under the domain model. The domain entities, under each attribute of the domain model, are instances or values of the attribute. For example, the entities *part-report, book-report* and *file-report* are the instances of the attribute *Report*. The domain entities in each row of the table is a domain instance. The three rows of domain entities are three domain instances, namely, *Stock-menu, Lib-menu* and *Arch-menu*. A domain model corresponds to a conceptual model in semantic data modelling. The attributes of a domain model define abstract concepts, and domain entities are the instances of the concepts.

In window *frame* a design framework is displayed, which consists of five interrelated objects: *library, book-report, lib-desk, reader-menu* and *manager.* The textual specification can be found in window *spec.*

In window *instance* a design instance, i.e. a particular *Lib-menu,* is displayed, which consists of five objects and the relationships between them. The objects are *library, book-report, lib-desk, reader-menu,* and *manager.* Comparing the objects appearing in the design instance and those of design framework, we see that the design instance is a specialized instance of the design framework. The design instance can be built by (1) instantiating the design framework with the domain instance *lib-menu,* producing an instance of the design framework, and (2) specializing the instance of the design framework by adding *manager,* a new object, to the instance of the design framework.

Figure 4.6.a is the top-level representation of the large-scale component *Lib-menu.* The objects as shown in the design instance of *Lib-menu,* can be refined into lower-level large-scale components. In order to see the refinement of an object one needs to click the node of the diagram which stands for the object to be refined. For example, figure 4.6.b can be displayed by clicking the node *lib-desk* of the design instance in figure 4.6.a.

The interface in figure 4.6.b is similar to that of figure 4.6.a. There are, however, some special nodes which are rectangular in shape in figure 4.6.b, such as *loan* and *return* in the design framework and *loan-A* and *return-C* in the design instance. A node which is rectangular in shape represents a primitive object. The specification of primitive objects can be found in window *spec*; the refinement of a primitive object is a source code component, which can be displayed if one clicks an object (a node of the diagram) of the design instance.

4.4.5 The process of creating and reusing large-scale components

The process of creating and reusing large-scale components is sketched in figure 4.7 and described as follows.

Creating large-scale components

The process of creating a large-scale component is concerned with the identification of an application domain, the creation of a

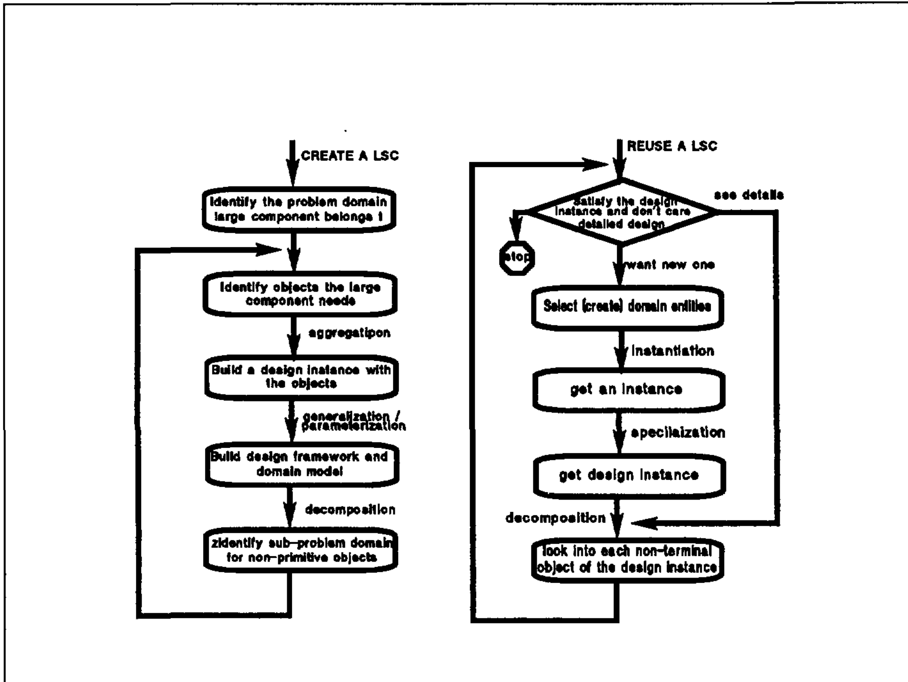


Figure 4.7. The process of creating and reusing large-scale components.

target system (or a large-grain component) in the domain, and generalization and parameterization of the target system with respect to the domain. Such a process can be largely described as follows:

1. *Identify the application domain concerned with a large-grain component.*
2. *Identify the objects of an application domain at an abstract level.*
3. *Build a design instance from the objects identified (aggregation).*
4. *Generate a design framework and domain resources by generalization and parameterization of the design instance.*
5. *Stop design process if all objects contained in the design instance are primitive.*
6. *Identify the sub-application domain represented by each non-primitive object in the design instance, then go to (2).*

Although aggregation and decomposition are alternatives for the creation of a large-scale component, aggregation is usually applied in object-oriented design as shown in the topology of large applications[20].

Reusing large-scale components

The process of reusing a large-scale component follows the H-structure of the component, dealing with the *composition process* at each level of abstraction and within the context of (1) a *design framework*, acting as an algorithm explaining how to organize lower level entities into alternative design instances, (2) a collection of *design instances*, each describing a particular design, (3) *domain resources* consisting of a domain model and a set of domain entities, and (4) a *refinement* as a detailed design and a particular implementation of a design instance. These four concerns are combined into an H-structure, encapsulating both a component's design and the context in which the design is realized, by which both the design information and the large-grain source code can be managed and manipulated to meet different design targets of an application domain.

A *composition process* refers to the process of building a design instance or an implementation of a system or a system component from existing artifacts which are available to be reused. The process of reusing a large-scale component can be largely described as follows:

1. **Verifying the design instance** *at an abstract level to see whether the existing design meets the requirement of the target object at this level. If yes, go to (5).*

2. **Selecting a domain instance** *from the domain resources or building a new domain instance. The latter can be done by selecting reusable entities from domain resources and/or creating new entities, if necessary. As a side effect, put the new domain instance into domain resources.*

3. **Instantiating the design framework** *with the domain instance in (2), resulting in an instance of the design frame work.*

4. **Specializing the instance** *in (3) into a design instance, and replacing the old design instance with the new one.*

5. **Stop** *if the design instance is complete and the designer satisfies the design (at an abstract level) and do not care about further details. Otherwise, looking into the refinement of the design instance,* **go to** *(1). A design instance is complete if it is a primitive object or if all the objects contained in the design instance are refined in terms of (lower-level) complete design instances.*

4.5 The capability of reusing large-scale components

The capabilities of reusing large-scale components are mainly reflected in two aspects: the capability of being reused and the capability of incremental domain analysis, which are described in this section respectively.

4.5.1 The life-cycle oriented reuse

The reusability of large-scale components is described in this subsection, concerning the stages of the improved waterfall model as discussed before. The stages are: analysis, design, implementation and maintenance.

Reusability information for analysis

Analysis is the process of modelling the world by identifying real world components which form the vocabulary of an application domain. Practically, each large-scale component (or its sub-component) models a concept from the real world, which is characterized by three dimensions: a design framework (a general concept), a design instance (an instance of the general concept), and a set of domain resources (sub-concepts and their thesauri), and these provide a vocabulary of an application domain.

Analysis represents the behavior of the system (or component) we must build. Such a behavior is contained in a large-scale component. The design framework of a large-scale component catches the common behaviors of a set of components of a domain, while the design instance describes the behavior of a specific component in terms of a set of objects and the relationships between them.

Reusability information for design

Design is the process of problem solving[112], which provides the abstractions and mechanisms that describe the behavior a system requires [20]. Large-scale components contain information to support this process.

A design framework provides a generic design by representing the commonalities of a set of design instances. Such a specification can be fully reapplied to build a collection of similar design instances in a domain and is, therefore, reusability information for design.

Although a design instance is horizontally specific with respect to the design framework at the same level, it is vertically a framework to be inherited by lower-level representations for varieties of design targets.

The domain resources are collected during the process of creating and reusing the large-scale component. They are listed in a large-scale component as candidate for the re-design or re-implementation of the constituent of a large-grain component, in order to reuse this component in different applications. Therefore, they contribute to the reuse of the design.

Reusability information for implementation

The major activity for system implementation is the construction of code components. Large-scale components not only contain code components but also provide a structure to integrate small code components into a large-grain component. Following such a structure, the large-grain component can be easily modified and reconstructed to implement varieties of target systems. An example of this will be provided in 7.

Actually, a large-scale component may be viewed as a mechanism in which small components are organized in such a way that reusability is emphasized. For example, in order to look up components in a software library, a domain model provides guidelines for the retrieval, while domain entities provide the components when needed. By identifying the relationships between a component and its related domain resources and design frameworks, the user may not only obtain the thesauri of the component, but also find a group of application environments where the component may be applied to. Therefore, the process of system implementation will be enhanced by using the information provided by large-scale components.

Reusability information for maintenance

It is generally accepted that the lack of design information and the lack of automated support in component manipulation complicate system maintenance.

Large-scale components support system maintenance by providing information which is useful for an easy modification and reconstruction of a subsystem or system (large-grain component) at different design levels. When a system is represented as a large-scale component, the design information of the system is distributed in a hierarchical structure. Information on this structure can be browsed with a tool. Any particular part of the system can be located when it needs to be modified, enhanced or reconstructed. Associated with each part, there is a template (design framework) which may be used to produce a similar design of the part, while furthermore there are domain resources which provide candidates to instantiate the template. Since the elements of domain resources may be other large-scale components, the detailed design and the code of the located part may be provided automatically.

4.5.2 Incremental domain analysis

Domain analysis, as first addressed by Neighbors[92] in 1980, refers to the activity of examining the requirements of a collection of systems that model a specific part of the real world. He thinks that domain analysis is only successful if it is done by a person who has built many systems for different clients in the same application domain[93].

Domain analysis is important as it is a basis for automating some aspects of system development in terms of reuse, such as application generators [92], domain-specific automatic programming [8] and so on. However, traditional domain analysis is done by *experts* and completed *before* using the analysis according to [92]. Such an analysis not only makes the initial use of the analysis very expensive, but also misses many opportunities to collect reusable resources from the process of system development.

In 1989, Arango[3] proposed that practical domain analysis should be based on a method for the systematic evolution of a reuser's model of the domain. In order to realize domain analysis, Arango also suggested a term *reuse infrastructure*, referring to the information and its structure which must be made available to the system developer, together with the auxiliary information needed to locate and manipulate this information.

We claim that large-scale components and the tools to manage and ma-

nipulate the components provide an infrastructure for reuse. While the design instances contained in a large-scale component are available to system developer to be used-as-is[10], the explicitly represented organizational structure provides means to localize the design instances at different levels of abstraction and to bind the instances into a whole system or a large-grain component. Moreover, the design framework and domain resources, corresponding to each design instance, provide the information to manipulate or reconstruct each part of the design instance to meet a reuser's requirements. Therefore, a large-scale component integrates information to be reused, and the information to realize the reuse.

We also claim that the process of reusing large-scale components supports incremental domain analysis. A large-scale component is created during the process of developing the first application system or large-grain component rather than performing domain analysis explicitly. The design information contained in a large-scale component can be incrementally extended during the reuse of the component, leading to an effective incremental domain analysis.

First, the design instance is a delegate of a set of similar designs in an application domain. Such a delegate can be replaced or modified into similar (but different) ones for different target systems or large-grain components in the same application domain. The new delegate is expected to be generated based on the domain resources and design framework. However, once the current domain entities in the domain resources are not sufficient to compose a new design instance, additional domain entities are needed. Reusers are responsible to supply these entities to enrich the current domain resources. Furthermore, if a reuser is not satisfied with the current design framework, a new one should be supplied. Replacing a design framework implies replacing a respected higher-level object with a new one. In this case the higher-level object together with its lower-level design will be put into the domain resources.

Obviously, the organizational structure of a large-scale component is a dynamic structure allowing the design information of an application domain to be incrementally accumulated. The more the component is reused, the more it will become reusable.

4.6 An information system for application reuse-in-the-large

According to title of this dissertation, as well, the discussion in previous sections, we address RITL, an information system for application **R**euse-**I**n-**T**he-**L**arge. Such an information system is centered on the representation and reapplication of large-scale components, leading to a support environment for application development and maintenance with reuse-in-the-large.

The data sets of the information system are the constituent parts of large-scale components as defined in section 4.2. The system software of the information system can be a semantic database management system as discussed in section 4.4. The people who use the information system are application developers. The major activities to deal with a large-scale component are described in terms of the type algebra as in section 4.3, including decomposition, aggregation parameterization, instantiation, generalization and specialization. The activity to collect and maintain the resources of a large-scale component is domain analysis as discussed in section 4.5.

The tools and languages to furnish the information system include:

1. a user interface, as discussed in 4.4.4, guiding application developers to browse and deal with large-scale components.

2. a specification language, specifying the constituent parts of a large-scale component,

3. graphic notations together with textual specification e.g. EDFG, representing the architecture of the design and the design framework of applications.

4. an implementation generator, providing support for composing large-grain source code components from lower-level entities, or delivering the large-grain source code components for an application,

5. a programming language, e.g. Scomp/Ocomp notations binding to C++, implementing code components or applications.

6. an editor, preparing specification and program.

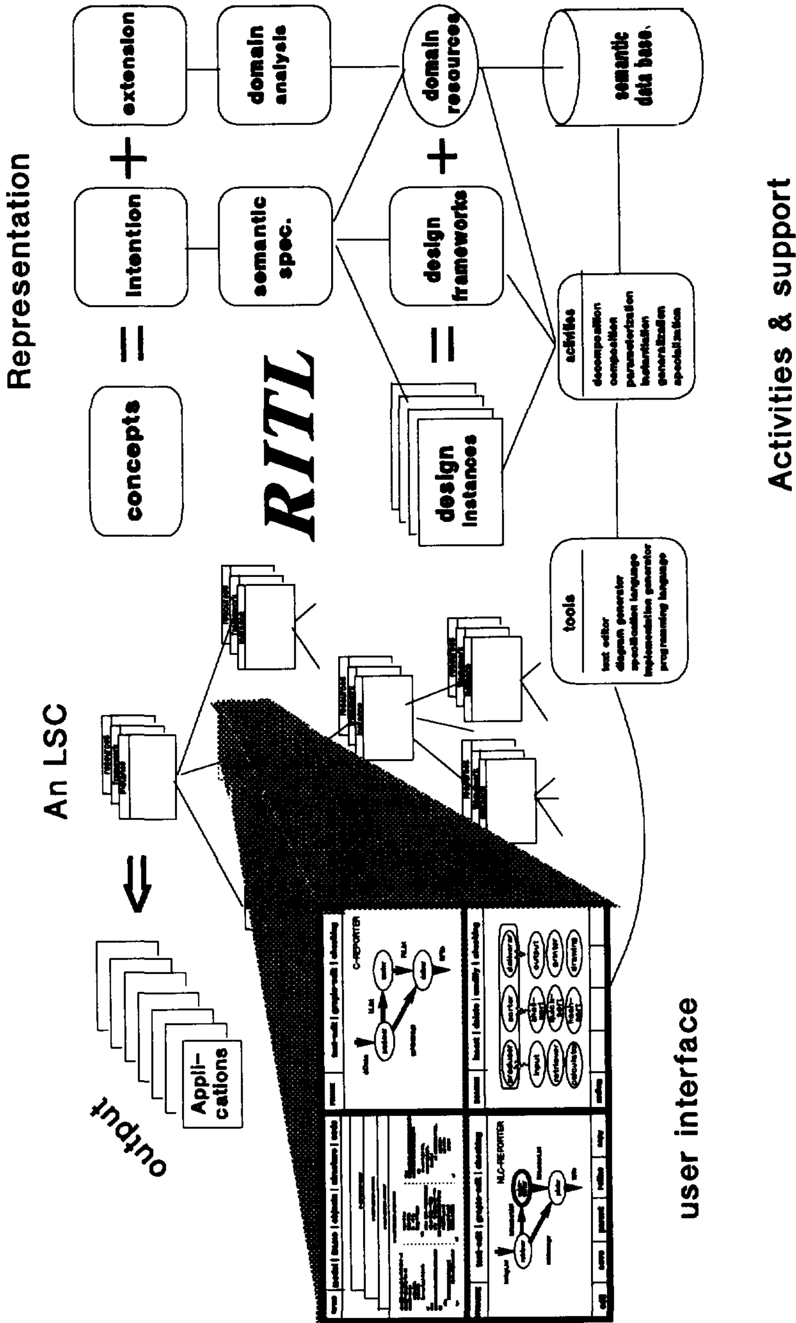


Figure 4.8. An Information system for application reuse-in-the-large

In figure 4.8, RITL is depicted, aiming at providing the logic structure of the information system, i.e. an outline of a support environment for application development and maintenance with reuse-in-the-large. The figure is focused on what information is represented in the information system, how it is presented, understood, processed, and applied to produce a set of applications in an application domain.

In the middle of figure 4.8, a large-scale component (An**LSC**) is presented in terms of an **H-structure**. Each node of the H-structure contains information, concerning a **design instance**, a **design framework** and **domain resources**. The nodes can be browsed and dealt with in terms of a **user interface** (left-bottom corner). At the mid-bottom of the figure, a set of **tools** are listed. The tools can be used to support a set of **activities** (besides the box tools). The activities are concerned with representing and manipulating **design instances, design frameworks** and **domain resources** (mid-right). Design instances can be composed from ('=') design frameworks and ('+') domain resources. The artifacts contained in the domain resources are collected in terms of **domain analysis**(mid-right) and managed in terms of a **semantic data base** (right bottom). The design instances, design frameworks and domain resources are specified in terms of a **specification language** (besides the box domain analysis). The specification describes the properties of artifacts, corresponding to the **intention**(right-top) of concepts; the domain analysis deals with the membership of the artifacts, corresponding to the **extension** (right-top corner) of concepts. **Concepts** (mid-top) can be expressed (=) in terms of intention and ('+') extension, which is helpful to understand things. Finally, as the **output** of the information system, a set of **applications** (left-top corner) can be derived from or developed with '<=') reusing a large-scale component.

4.7 Summary

In this chapter we described a method about how a large-scale component can be represented, managed and manipulated. At the beginning, we identified several notations which form a basis to describe the method. Then, the representation, manipulation and management of a large-scale component were described from different views in terms of an operating model and a man-

agement model. Moreover, component specification was discussed. Finally, we discussed the specification of large-scale components, binding to a module interconnection language.

Based on the discussions above, several conclusions can be drawn:

1. Reuse-in-the-large — the reuse of large-grain components and the information about the creation of the components — can be realized by the reuse of large-scale components which contains design instances, design frameworks and domain resources, representing different levels of abstraction.

2. The management and manipulation of a large-scale component can be supported by the technologies applied in semantic data modelling, leading to a component base, supporting the process of reuse-in-the-large.

3. The reuse of large-scale components supports incremental domain analysis, and is capable of improving system development and maintenance.

Chapter 5

The Specification of Large-Scale Components

Specification languages usually insist upon specifying the details precisely or not at all. But we cannot specify the details too precisely, else we reduce the reuse potential.

— Ted J. Biggerstaff, *Software Reusability,* 1989.

In the previous chapter we discussed the management of large-scale components by assuming that large-scale components are represented in terms of primitive code components and specifications. In this chapter we discuss a principle for such specifications and describe how to apply such a principle for the representation of large-scale components. Moreover, we address a method for consistency verification.

5.1 The specification of large-scale components

The specification of a large-scale component is the design representation of the component. In order to enhance the reusability of such a representation, *multiple-level specification* is used.

Multiple-level specification was first addressed in 1991 by Gabrielian[44] for specifying real-time systems. He distinguishes multiple-level specification from hierarchical specification. Hierarchical specification usually begins with a top-level specification. Thereafter, one creates successively lower-level inde-

pendent specifications that are proved to be consistent with higher-levels. At the end of the process, the lowest level stands alone as a final specification. In contrast, multiple-level specification allows each higher-level specification to impose constraints on lower-level specifications, so that all levels remain part of the final specification[44]. An important feature of multiple-level specification is that each level can be **reused** in other contexts.

The multiple-level specification in our approach is realized in two dimensions: vertically, in the relationships between different levels of refinement, and horizontally, in the relationships between domain model, design framework and design instance.

Vertically, the specification of a design instance must inherit the specification of its parent object, if any, and the specification of an object appearing in the design instance must be inherited by the specification of its refinement, if any. Horizontally, the specification of a design instance may inherit the specification of the design framework, and the specification of the design framework must inherit the specification of the domain model.

In terms of multiple-level specification, the specification of system components can be specified in different levels of abstraction. Each level is a *partial specification*, and a complete specification can be finally obtained at the lowest level in terms of inheritance. A specification is called partial specification if it describes the broad structure of an object and leaves the details incomplete, i.e. only minimally constrained[18].

In this section we describe the principle of the multiple-level specification and how to apply such a principle to the specification of large-scale components.

5.1.1 Multiple-level Specification

In this subsection we provide a definition of multiple-level specification as well as the *refinement consistency* regarding multiple-level specification. Then we discuss the reusability of such kind of specification.

What is multiple-level specification?

Assuming that the specification of a system component, say A, can be represented in terms of a conjunction of a set of predicates in first-order logic, the

complete specification of A can be represented as follows:

$$SPEC(A) = P1(A) \wedge P2(A) \wedge ... \wedge Pn(A) \qquad (*)$$

where *SPEC(A)* refers to the specification of a component A, $P_i(A)$ $(i = 1, 2, ...n)$ is a predicate regarding the component A. In contrast, the multiple-level specification of component A can be

level 1	$SPEC_1(A)=P1(A)$
level 2	$SPEC_2(A)=P2(A)$
...	...
level n	$SPEC_n(A)=Pn(A)$

For the specification above we further stipulate that each lower-level specification inherits the specification of higher-level ones, then the specification above is equivalent to a description logically as follows:

level 1	$SPEC_1(A)=P1(A)$
level 2	$SPEC_2(A)=P1(A)\wedge P2(A)$
...	...
level n	$SPEC_n(A)=P1(A) \wedge P2(A)\wedge ... \wedge Pn(A)$

From the representation above, we see that the lowest-level specification is (logically) equivalent to that of an original specification (∗), without violating the principle of multiple-level specification.

Refinement consistency

The consistency between lower level and higher level specifications, namely, refinement consistency, is defined as follows:

definition 3 *(refinement consistency) Refinement consistency is valid by a multiple-level specification iff*

$$SPEC_i(A) \Longrightarrow SPEC_{i-1}(A)$$

for i=2,3,...n.

Consequently, assuming that *set-A_i (i=2, 3, ..., n)* is a set of components satisfying the specification *$SPEC_i(A)$*, then

$$set\text{-}A_i \subseteq set\text{-}A_{i-1}$$

As a result, the lowest level specification $SPEC_n(A)$ presents the complete specification of component A, and each $SPEC_1(A)$, $SPEC_2(A)$, ..., $SPEC_n(A)$ are partial specifications.

In order to understand the definition above more precisely, it seems necessary to distinguish the refinement of multiple-level specification from that of hierarchical specification. An example is provided in figure 5.1.

Items	Hierarchical	Multiple-level
parent spec.	function f(x) return y pre(f): $x \in Integer \wedge x \geq 0$ post(f): $y = \sqrt{x}$	function f(x) return y pre(f): $x \in Number$ post(f): $y = \sqrt{x}$
refinement	function $f'(x)$ return y pre(f′): $x \in Number \wedge x \geq 0$ post(f′): $y = \sqrt{x}$	function f′(x) return y pre(f′): $x \in Integer \wedge x \geq 0$ post(f′): $y = \sqrt{x} \wedge y \leq 100$
consistency	pre(f) $\Longrightarrow$ pre(f′) post(f′) $\Longrightarrow$ post(f)	pre(f′) $\Longrightarrow$ pre(f) post(f′) $\Longrightarrow$ post(f)

Figure 5.1. Hierarchical Refinement versus Multiple-level Refinement.

In figure 5.1 we assume that $\forall x(x \in Integer \Longrightarrow Number)$. Notice that the precondition must be equivalent or *weaker* than that of the parent specification in hierarchical refinement, whereas it must be equivalent or *stronger* than that of parent specification in multiple-level refinement.

The reusability of multiple-level specification

We present the reusability of multiple-level specification in terms of an example as follows,

level 1	BUILDING(A):	A is an object with wall and roof.
level 2	HOUSE(A):	A is a BUILDING which is for people to live in.
level 3	B-HOUSE(A):	A is a HOUSE with bungalow style.
...	...	...

where BUILDING(A), HOUSE(A), and B-HOUSE(A) are predicates of object A. The example above is intended to specify a bungalow style house. The specification is provided with three levels of specification. At the first level the basic structure of a house is specified, at the second level the function of a house is provided, and at the third level the style of a house is given.

From the example above, we can see intuitively that the refinement consistency is held in the specification. If an object is a B-HOUSE, it must be a HOUSE; if an object is a HOUSE, it must be a BUILDING, i.e.

$$\forall x \in Object(B\text{-}HOUSE(x) \Longrightarrow HOUSE(x)) \wedge$$
$$\forall y \in Object(HOUSE(y) \Longrightarrow BUILDING(y))$$

where *Object* refers to all objects which are considered in an application domain. Moreover, assuming set-B-HOUSE presents all objects which satisfy the specification at level 3, set-HOUSE presents all objects which satisfy the specification at level 2, and set-BUILDING presents all objects which satisfy the specification at level 1, the following expression must be held:

$$\text{set-B-HOUSE} \subseteq \text{set-HOUSE} \subseteq \text{set-BUILDING}$$

The reusability of the specification above is obvious. On the one hand, each higher-level specification can be reused by the lower-level specification: the level 1 can be reused for the specification of alternative buildings, the levels 1 and 2 together can be reused for the specification of alternative houses and the levels 1, 2, and 3 can be reused for the specification of alternative houses with bungalow style; on the other hand, the lower-level specifications provide instances of the higher-level specification conceptually: B-HOUSE is an instance of HOUSE and HOUSE is an instance of BUILDING. Moreover, as the specification is divided into several levels, the specification can be understood and manipulated easily. This follows the strategy "divide and conquer", an old saying.

5.1.2 Specifying large-scale components

In this subsection we discuss how multiple-level specification can be applied to the specification of large-scale components. The discussion will be focused on how to specify design frameworks, domain models, design instances and the levels of refinement. The feature of multiple-level specification is the inheritance between different levels of specification. For convenience's sake, the notations in EDFG (extended data flow graph)[74], a module interconnection language, are frequently used.

Specifying design frameworks

Design frameworks represent the organizational structure of the objects at each level of abstraction. The organizational structure corresponds to the composition of a higher-level object from its sub-objects. The complexity of such a composition can be reduced by hiding the complex control structures and data structures inside the sub-objects, and leaving a unified control structure to connect the sub-objects into a whole. The well-known methods supporting such an idea include Unix pipeline, data flow graph, the messages between objects in object-oriented programming, and so on.

For the same reason the specification of a design framework is limited to the relationships between the sub-objects to be composed with, concerning a unified control structure. We assume that a set of sub-objects can be composed into a single component by using algorithm structures or system architectures.

The languages representing a design framework are typically MILs[99] which provide a specification of component structures. The specification can be precise if a semantics is provided. As an example, we use semantically extended data flow graphs (EDFG[74]) for the specification, see also section 5.2. The question seems how data flow graphs can be used to precisely describe the interconnections between actors.

We argue that data flow graphs can be used to describe precisely the interconnections between objects by a semantic extension, i.e. by providing a textual specification as we did in the EDFG approach[74]. Some experience for such a capability has also been given by Larsen's work[61] and France's report[40]. In the former data flow graphs extended with pre/post style semantic specification can be directly transformed into VDM specification; in the latter both data flows and control flows of a system can be represented in terms of semantically extended data flow graphs. Moreover, the consistency

between a semantically extended data flow graph and its refinement can be verified in a rigorous manner, which will be discussed later on.

The specification of a design framework is a partial specification. Although the specification of a design framework describes the general structure of a set of similar design instances in an application domain, such a specification may provide no semantic constraints on the objects appearing in the design framework. These constraints will be provided by a domain model.

Specifying domain models

A domain model describes the constraints of the (abstract) objects appearing in a design framework. These constraints can be provided in terms of a specification. The specification is characterized by semantics with abstraction. The semantics with abstraction can be used to identify a collection of resources for the instantiation of the design frameworks.

The specification of a domain model should meet the constraints imposed on the design framework. For example, if two objects appear in the design framework as being connected, the specification of these two objects in the domain model should express the possibility of the required connection. In other words, we may stipulate that one can only connect two objects in a design framework if the interconnection is consistent with the domain specification. The consistency of such an interconnection should be verified, based on the specification of the two components.

In order to emphasize the abstraction of the objects appearing in a design framework and to identify resources for the instantiation of the design framework, the specification of the domain model emphasizes only necessary properties of the system components and ignores the others, so that the same specification catches the commonality of a set of similar components and can be shared by all the components. Following the abstract specification, a collection of domain entities can be identified as thesauri, which can be reused to instantiate the design framework. For example, *sort*, as an attribute of a domain model, can be specified as an abstract object: its input includes a set of data items with the same type in which an total order is defined, and its output is a set of sorted data items. The domain entities which can be identified by such a specification may include quick-sort, hashing-sort, tree-sort and so on. These domain entities are so-called thesauri regarding to *sort*.

Specifying design instances

According to the definition of a large-scale component, the specification of a design instance requires three activities: (1) describing the architectural structure of the design instance; (2) defining the objects appearing in the design instance; (3) specifying the external interface.

Specifying the objects appearing in a design instance. Specifying the objects appearing in a design instance can be viewed as an extension of the corresponding domain model. This is because most of the objects appearing in a design instance are the domain entities of the domain resources and the domain entities are partially specified by the domain model at an abstract level. By an extension of the domain model, only some particular properties of the objects are need to be specified. This is a kind of reuse. However, the extension may not be available for each object, since a design instance can be an instance of the design framework and can also be the specialization of the instance. The latter may need some particular objects which are not covered by the corresponding domain model.

Specifying the organizational structure of a design instance. The same holds for the specification of a design framework. An organizational structure of a design instance can be specified in terms of a module interconnection language. For specifying such a structure, the specification of the design framework can be reused because the organizational structure of a design instance is but a more specific design framework, or the specialization of the design framework. For example, assume that the data type of a link from an abstract actor A1 to A2 in the design framework is a union List, List={real-List, integer-List, ...}, the data type of the corresponding link in the design instance may be a particular link, say real-List. Of course, it is necessary to specify additional structures when the design instance is a specialized instance of the design framework. From this point of view, the specification of design framework provides a partial specification of the design instance.

Specifying external interface. The external interface of a design instance can be specified in a model interconnection language, e.g. the EDFG notation. If the design instance is at the top-level of a large-scale component, the interface can be specified straight forward; if the design instance is at a lower-level of a large-scale component, the specification of its external interface should be consistent with that of higher-level specification. The consistency implies two

things. On the one hand, the specification at the lower-level should inherit all the specifications at the higher-level; on the other hand, the specification of a lower-level design instance should be consistent with the specification of another design instance which the current design instance has to be connected with. For example, suppose there is a design instance at top-level, see also figure 5.2, which consists of two actor A1 and A2. A1 has an input link l01 and an output link l02; A2 has an input link l02 and an output link l03; A1 and A2 are connected in terms of l02. Suppose there are edfg1 and edfg2 which are the refinements of actors A1 and A2 respectively. The external input links of edfg1 are l01 and l02 and the external output links of edfg2 are l02 and l03, corresponding to the higher-level actors A1 and A2. Then the specification of the four objects A1, A2, edfg1 and edfg2 can be formalized as

PRE(A1)=P_{a1}(l01)	PRE(A2)=$P3_{a2}$(l02)
POST(A1)=P_{a2}(l02)	POST(A2)=$P4_{a2}$(l03)
PRE(edfg1)=P_{e1}(l01)	PRE(edfg2)=P_{e2}(l02)
POST(edfg1)=P_{e1}(l02)	POST(edfg2)=P_{e2}(l03)

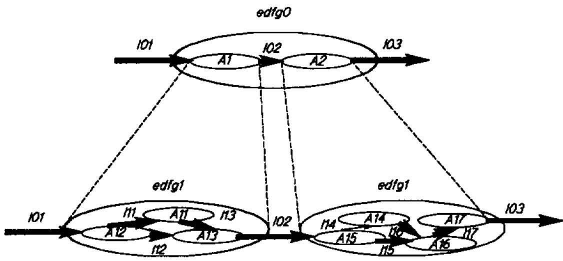


Figure 5.2. The refinement of a design instance.

where P_{xj}(lj) is a predicate on link lj, presenting the pre/post condition of an object xj. In this case the following predicate must be true:

$$P_{e1}(l01) \Longrightarrow P_{a1}(l01) \wedge P_{e1}(l02) \Longrightarrow P_{a1}(l02) \wedge$$
$$P_{e2}(l02) \Longrightarrow P_{a2}(l02) \wedge P_{e2}(l03) \Longrightarrow P_{a2}(l03) \wedge$$
$$P_{e1}(l02) \Longrightarrow P_{e2}(l02)$$

The boolean expression $P_{e1}(l01) \Longrightarrow P_{a1}(l01) \wedge P_{e1}(l02) \Longrightarrow P_{a1}(l02)$ must be true. This implies that the links 101 and 102, the external links of *edfg1*, must be the instances of the (abstract) links with the same names appearing in A1, or be the same to the links in A1. For the similar reason, the boolean expression $P_{e2}(l02) \Longrightarrow P_{a2}(l02) \wedge P_{e2}(l03) \Longrightarrow P_{a2}(l03)$ must be true as well. Finally, the boolean expression $P_{e1}(l02) \Longrightarrow P_{e2}(l02)$ must be true since the connection is necessary between *edfg1* and edfg2 through link *l02* as indicated in the higher-level specification.

5.2 An example of large-scale components

As an example, we specify *data processor* (or *dataproc)*, a large-scale component, with a purpose of the production and sorting of data. Assume that the general structure of the component is as a triple with a *producer, sorter* and *deliverer.* The component is parameterized on the objects of the triple. These objects, therefore, appear in the domain model. The general structure of the large-scale component is described by the design framework, while an instance of the example is given in the design instance. For convenience's sake, such an example is specified in the EDFG notation [74].

The top-level representation of the large-scale components. In the example, see also figure 5.3, the domain model at the top-level of the large-scale component consists of three attributes which are specified in terms of pre/post conditions. Under each attribute of the domain model, several objects, i.e. domain entities, are listed. As a rule, the specification of each domain entity must be consistent with the specification of the corresponding attribute. Each set of entities at the same row, which is called a *domain instance*, is sufficient to instantiate the design framework so as to build a design instance.

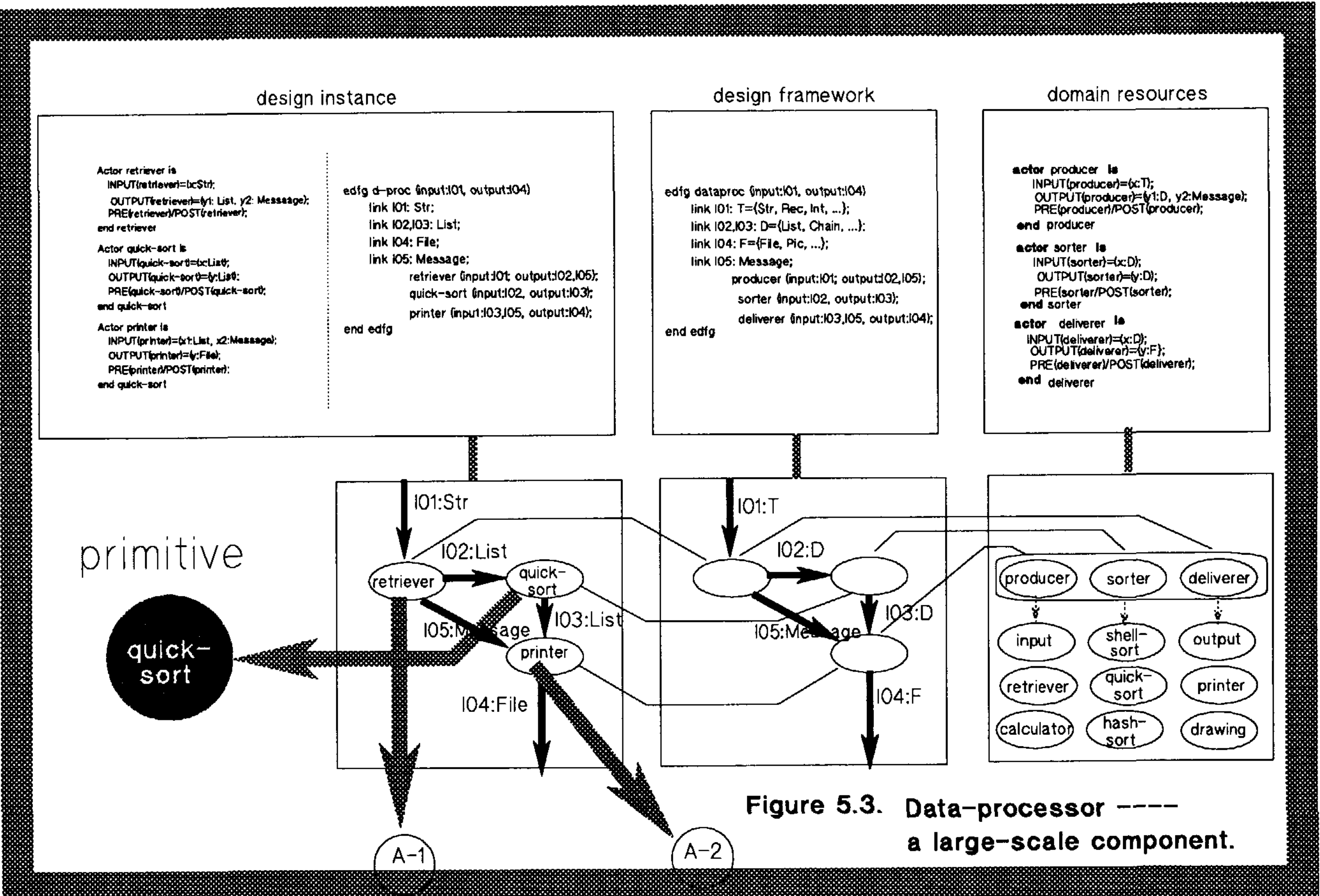
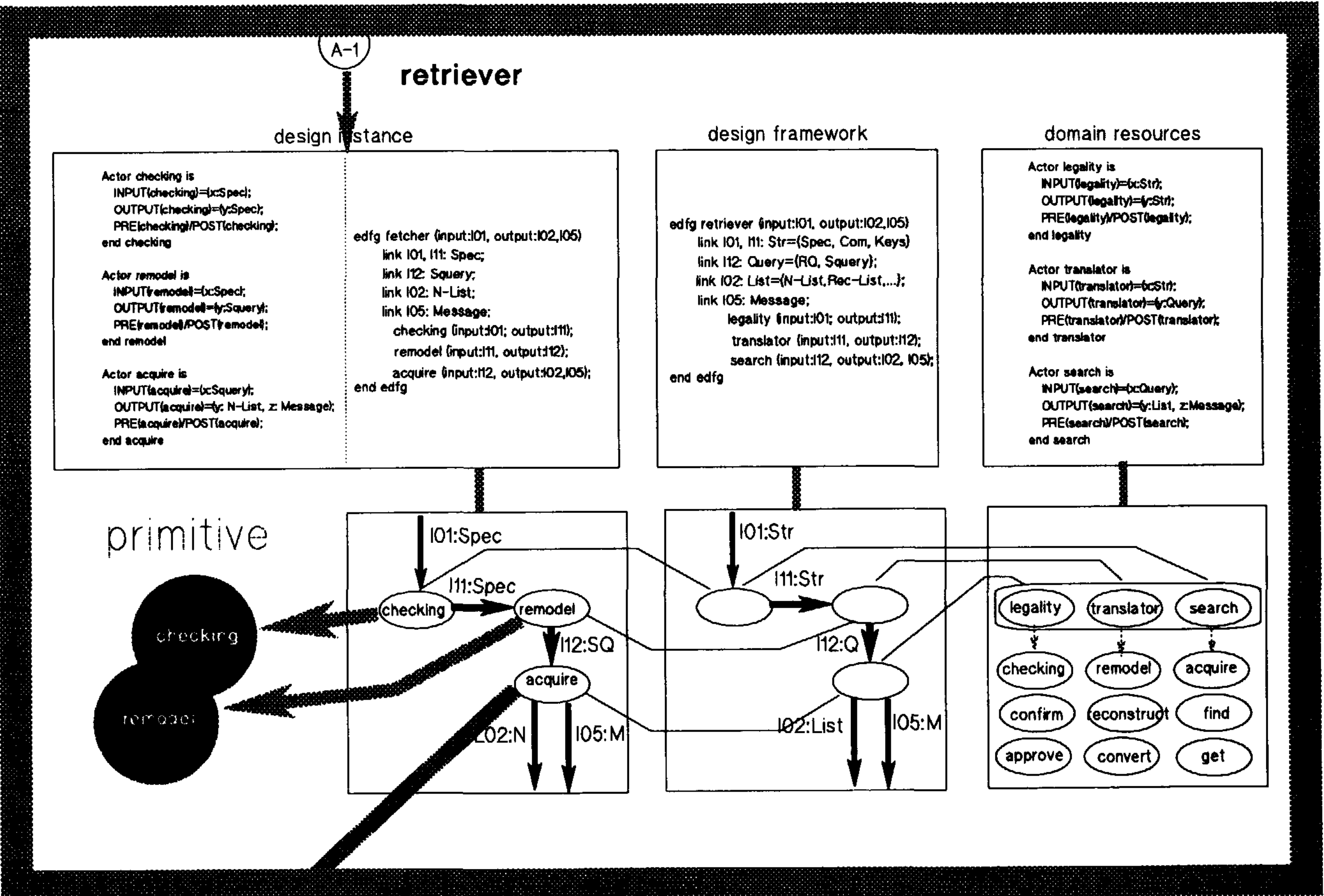
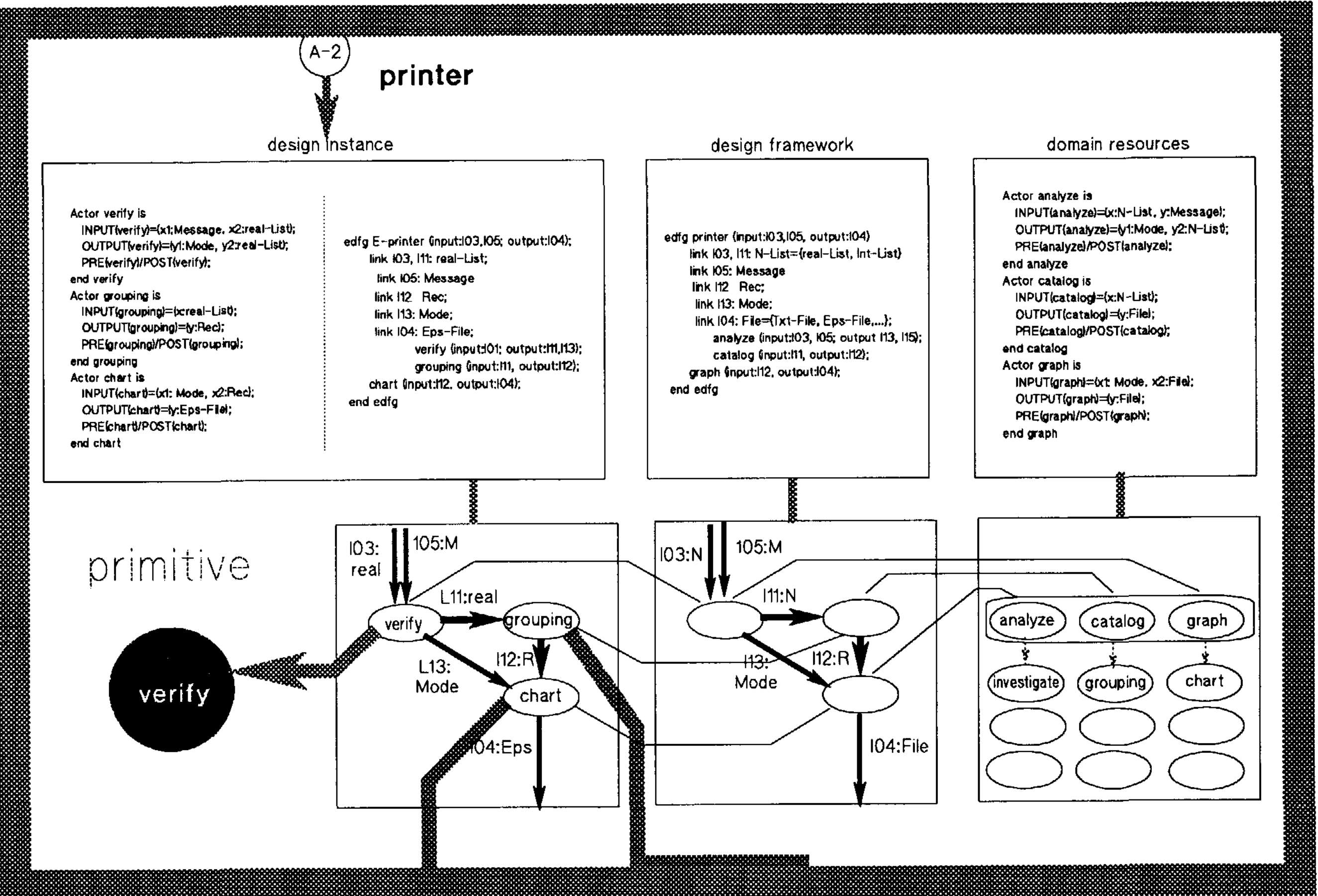


Figure 5.3. Data-processor ---- a large-scale component.



A-1
retriever
design instance
design framework
domain resources
Actor checking is
INPUT(checking)=(x:Spec);
OUTPUT(checking)=(y:Spec);
PRE(checking)/POST(checking);
end checking
Actor remodel is
INPUT(remodel)=(x:Spec);
OUTPUT(remodel)=(y:Squery);
PRE(remodel)/POST(remodel);
end remodel
Actor acquire is
INPUT(acquire)=(x:Squery);
OUTPUT(acquire)=(y: N-List, z: Message);
PRE(acquire)/POST(acquire);
end acquire
edfg fetcher (input:I01, output:I02,I05)
link I01, I11: Spec;
link I12: Squery;
link I02: N-List;
link I05: Message;
checking (input:I01; output:I11);
remodel (input:I11, output:I12);
acquire (input:I12, output:I02,I05);
end edfg
edfg retriever (input:I01, output:I02,I05)
link I01, I11: Str=(Spec, Com, Keys)
link I12: Query=(RQ, Squery);
link I02: List=(N-List,Rec-List,...);
link I05: Message;
legality (input:I01; output:I11);
translator (input:I11, output:I12);
search (input:I12, output:I02, I05);
end edfg
Actor legality is
INPUT(legality)=(x:Str);
OUTPUT(legality)=(y:Str);
PRE(legality)/POST(legality);
end legality
Actor translator is
INPUT(translator)=(x:Str);
OUTPUT(translator)=(y:Query);
PRE(translator)/POST(translator);
end translator
Actor search is
INPUT(search)=(x:Query);
OUTPUT(search)=(y:List, z:Message);
PRE(search)/POST(search);
end search
primitive
checking
I01:Spec
I11:Spec
checking
remodel
I12:SQ
acquire
I05:M
I01:Str
I11:Str
I12:Q
I02:List
I05:M
legality
translator
search
checking
remodel
acquire
confirm
find
approve
convert
get


A-2
printer
design instance
Actor verify is
INPUT(verify)=(x1:Message, x2:real-List);
OUTPUT(verify)=(y1:Mode, y2:real-List);
PRE(verify)/POST(verify);
end verify
Actor grouping is
INPUT(grouping)=(x:real-List);
OUTPUT(grouping)=(y:Rec);
PRE(grouping)/POST(grouping);
end grouping
Actor chart is
INPUT(chart)=(x1: Mode, x2:Rec);
OUTPUT(chart)=(y:Eps-File);
PRE(chart)/POST(chart);
end chart
edfg E-printer (input:I03,I05; output:I04);
link I03, I11: real-List;
link I05: Message
link I12 Rec;
link I13: Mode;
link I04: Eps-File;
verify (input:I01; output:I11,I13);
grouping (input:I11, output:I12);
chart (input:I12, output:I04);
end edfg
design framework
edfg printer (input:I03,I05, output:I04)
link I03, I11: N-List={real-List, Int-List}
link I05: Message
link I12 Rec;
link I13: Mode;
link I04: File={Txt-File, Eps-File,...};
analyze (input:I03, I05; output I13, I15);
catalog (input:I11, output:I12);
graph (input:I12, output:I04);
end edfg
domain resources
Actor analyze is
INPUT(analyze)=(x:N-List, y:Message);
OUTPUT(analyze)=(y1:Mode, y2:N-List);
PRE(analyze)/POST(analyze);
end analyze
Actor catalog is
INPUT(catalog)=(x:N-List);
OUTPUT(catalog)=(y:File);
PRE(catalog)/POST(catalog);
end catalog
Actor graph is
INPUT(graph)=(x1 Mode, x2:File);
OUTPUT(graph)=(y:File);
PRE(graph)/POST(graph);
end graph
primitive
verify
I03: real
105:M
L11:real
verify
grouping
L13: Mode
I12:R
chart
I04:Eps
I03:N
105:M
I11:N
I13: Mode
I12:R
I04:File
analyze
catalog
graph
investigate
grouping
chart

dataproc, the design framework at the top-level of the large-scale component, describes the organizational structure of the components in terms of the attributes of the domain model. In this example the structure is described in two parts, a textual part and a graphic part. The textual part describes the interface of the objects (actors). The graphic part explicitly shows the interconnections between the objects. Two objects are connected if the input link of one object is the output link of the other one.

d-proc, the design instance at the top-level of the large-scale component represents an instantiated component in this example, which consists of *retriever, quick-sorter,* and *printer.*

The refinement. Although *d-proc*, the design instance at the top-level of the large-scale component, is a particular instance of the design framework of this level, it is too abstract to act as a specific *data-processor.* For example, the link type *List* still has to be supplied as shown in the type tree of figure 5.3a.

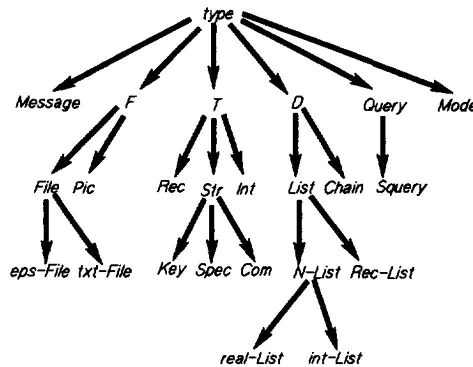


Figure 5.3a. A type tree.

The lower-level representation of the large-scale component is concerned with the three objects appearing in *d-proc*, i.e. *retriever, quick-sort* and *printer.* Since *quick-sort* is a primitive object, it is connected with a source code component which can be accessed to be used-as-is and no more refinement is necessary. The refinements of *retriever* and *printer*, two non-primitive objects, are shown in (A-1) and (A-2), the remaining part (in other two pages) of figure 5.3.

The refinement of *retriever.* The refinement of *retriever* is given by a design framework with the same name, as shown in (A-1), which suggests that *retriever* basically consists of three objects: *legality, translator* and *search.* The

legality receives a requirement description from link *l01:Str* and checks if such a description is legally right. After the checking, it sends a correct requirement description via link *l11:Str* to *translator* which transforms the requirement description into query notations of a database. The query notations are sent to *search*, another object, via link *l12:Query*. The *search* realizes retrieval in terms of the query notations, resulting in a list of data and a message about the data. The data and message are sent out via links *l05* and *l*02 respectively.

The objects *legality, translator* and *search* appearing in the design framework *retriever*, are still abstract objects. Their constraints are described in the domain model regarding the design framework. Suppose that domain entities have been collected as many as shown in (A-1), a set of alternative *retrievers* can be composed by instantiating the design framework with the entities.

As a delegate of all possible *retrievers*, *fetcher* presents the design instance. The correspondence between the design framework and the design instance is as follows:

Design-framework & Design-instance

retriever $\mapsto$ fetcher: a particular retriever.
legality $\mapsto$ checking: a particular legality.
translator $\mapsto$ remodel: a particular translator.
search $\mapsto$ acquire: a particular search.
Str $\mapsto$ Spec: a subtype of Str.
Query $\mapsto$ Squery: a subtype of Query.
List $\mapsto$ N-List: a subtype of List.
Message $\mapsto$ Message: a subtype of Message.

where "$\mapsto$" refers to the correspondence between an abstract object and one of its instance.

When a design instance is created, the **refinement consistency** must be verified with respect to the higher-level design instance. Viewing the design instance as a box, we only need to pay attention to the external links of this design instance. In this example, the external links include *l01, l02* and *l05*. If the refinement consistency is held, the following boolean expressions must be true.

One-level refinement consistency

$$PRE_{l01}(checking) \Longrightarrow PRE_{l01}(retriever) \wedge$$
$$POST_{l02,l05}(acquire) \Longrightarrow POST_{l02,l05}(retriever) \wedge$$
$$POST_{l02}(acquire) \Longrightarrow PRE_{l02}(quick\text{-}sort) \wedge$$
$$POST_{l05}(acquire) \Longrightarrow PRE_{l05}(analyze)$$

Note that we use "$\Longrightarrow$ (implies)" in the table above, rather than "$\Longleftarrow$ (be implied)" and "$\Longleftrightarrow$ (is equivalent to)", which reflects the feature of multiple-level specification as discussed before.

In the table above the object *analyze* appearing in $PRE_{l05}(analyze)$ refers to an object in the refinement of *printer*, see also (A-2), the third part of figure 5.3.

The refinement of ***printer.*** The third part of figure 5.3 shows the refinement of *printer.* The process of such a refinement is roughly similar to that of the refinement for *retriever.* However, there is only one domain instance contained in the domain resources. If a reuser does not satisfy the current design instance for its sub-objects, new domain entities should be provided directly. In this way, new domain resources are accumulated. This is important, for only by such an accumulation an incremental domain analysis can be realized, so that the reusability of *data-processor* can continually be increased.

Finally, although the example in this subsection is only described partially, we suppose that the rest part, the further lower-level refinement of the large-scale component, can be completed by readers since a large-scale component is recursively defined. Additionally, the notations used for the specification are only used for an example, as we believe that the general form of large-scale components is language independent.

5.3 The consistency verification

In the previous sections we discussed how large-scale components are specified and represented in terms of different levels of abstraction. In this section a method is presented for consistency verification between the different levels, especially between a high level object and its refinement. This method suggests

that the verification can be simplified by inference from the extended data-flow graphs with a few rules.

In chapter 3 an EDFG approach was presented as a basis for the reuse of system design. According to this approach an application system can be represented in terms of semantically extended data flow graphs (*edfg*), a kind of model interconnection language. If an application system or a large-grain component is represented in terms of different levels of abstraction with *edfg* notation, it can easily be maintained and reconstructed to meet the requirements for varieties of design targets.

In chapter 4 the structure of different levels of abstraction — which are described in chapter 3 — is extended into a general structure, the H-structure, for the representation of large-scale components. In this case, the *edfg* notation, which was used in representing system designs, can also be the basic notation for the specification of large-scale components.

Since *edfgs* are semantically extended data flow graphs, the semantic extension should lead to a possibility to rigorously verify refinement consistency. In this section we will discuss what we mean by refinement consistency and how to verify the consistency.

The techniques applied in our method for consistency verification are drawn from France's method for refinement verification[40] and Hoare's notation on correctness proof[49]. Like France's method, a black-box approach is used for consistency verification. Like Hoare's notation a set of inference rules are used for the verification.

The result is a method, *edfg-merging*, by which a complex *edfg* can be simplified with a few inference rules, without loosing its properties in order to verify consistency. In this case the consistency verification of a refinement with respect to its parent actor amounts to the verification of a predicate.

5.3.1 The concepts of actor and *edfg*

The concepts of actor and *edfg* were described for the representation of system design in chapter 3. They are discussed again in this subsection, in order to make clear the needs for the specification of large-scale components. The discussion from now on will follow the description in this subsection. As discussed in chapter 3, a system component can be modeled in terms of an actor and an *edfg*. An actor is a system component which consists of an agent for application processing and links for the input and output of the agent. An

edfg consists of a set of interconnected actors. Two actors are interconnected if at least one output link of one actor is an input link of the other actor. The relationships between actors and *edfgs* are that: an actor can be an abstraction of an *edfg*; an *edfg* can be a refinement of an actor and contains a set of other actors.

An actor or an *edfg* can be described in two forms, i.e. graphically or textually. A *graphical actor*, see also figure 5.4, consists of a box and links. The box represents the application agent of the actor, which is labeled with the name of the actor. The links coming to the agent are the *input links* of the actor, and the links leaving the agent are the *output links* of the actor. Each link is labeled with a link name. Moreover, the identifiers of precondition and postcondition of the actor are attached to the box at the sides of input links and output links respectively.

A *graphical edfg*, see also the example in figure 5.4, is a group of graphical actors. Each actor of this group must at least have one input link coming from (or one output link going to) another actor of the same group. A link, which is both an output link of one actor and an input link of another actor of an *edfg*, is called an *internal link* of the *edfg*, otherwise an *external link.*

A *textual actor* is a 5-tuple: *actor-name, input links, output links, precondition and postcondition*, which can be generally represented as follows:

actor a **is**
$INPUT(a)=(l_1 : d_1, l_2 : d_2, ...l_n : d_n)$
$OUTPUT(a)=(l_{n+1} : d_{n+1}, l_{n+2} : d_{n+2}, ...l_{n+k} : d_{n+k})$
$PRE(a)$: $INPUT(a) \longmapsto Boolean$
$POST(a)$: $INPUT(a), OUTPUT(a) \longmapsto Boolean$
end a

where a is the name of an actor. *INPUT(a)/OUTPUT(a)* define input/output links of the actor, corresponding to *IFS(a) and OFS(a)* in chapter 3. A link can be either a *queued link* that behaves like a queue, or a *state link* that behaves like a program variable [40]. *PRE(a)/POST(a)* define the pre/postcondition of the actor. A pre/post condition is a predicate or assertion on the input/output links. The notation in VDM[54] can be adopted for such definitions. Moreover, in contrast with the PRE(a) in chapter 3, the same notation in this section is short for the conjunction of PRE(a) and FIRE(a) in chapter 3. Additionally, both the signatures $INPUT(a) \mapsto Boolean$ and $OUTPUT(a) \mapsto Boolean$ indicate the necessity of a kind of predicates and should be replaced in practice.

The textual *edfg*, see also the example in figure 5.4, contains a series of actor specifications and a structure description, the former providing the syntactic and semantic description of the actors, the latter describing the *interconnections between the actors.* The interconnection between actors can be described by providing the signature of each actor and definitions of all links required. Two actors are interconnected if the same link appears in each of their signatures.

Additionally, the internal and external links should be distinguishable and a distinction must be made between queued links and state links.

The relationships between graphic *edfgs* and textual *edfgs* are, that the graphical *edfg* provides an intuitive view of the textual *edfg*, and that the textual *edfg* provides a precise definition and a semantic extension of the graphical *edfg*.

The purpose of using *edfg* is to represent a system design with different levels of abstraction. The relationships between the different levels is that an actor appearing in a higher level *edfg* can be refined in terms of a lower level *edfg*. In order to emphasize such a relationship, the basic system component, see also figure 5.4, is represented in terms of a specification part, i.e. **actor spec**, and a refinement part, i.e **actor body**. The former is simply an actor specification, the latter packages an *edfg* and provides the definitions of all actors appearing in the *edfg*.

Restriction. An *actor* is either a *data transform* or a *state bearing object.* The output of a data transform depends solely on its input, whereas a state bearing object may also depend on past input. However, our discussion in this section will be restricted to data transform because, as indicated by France[40], processing objects whose output also depends on past input can be modeled as data transforms communicating with a data store that stores the effects of past execution. Such a data store can be modeled with state links in our approach.

5.3.2 The refinement consistency

Consistency of refinement in our approach implies that the specification of a detailed design or implementation of an actor is capable of meeting the (requirement) specification of the actor. Such a consistency refers to both syntactic consistency and semantic consistency, which are defined as follows:

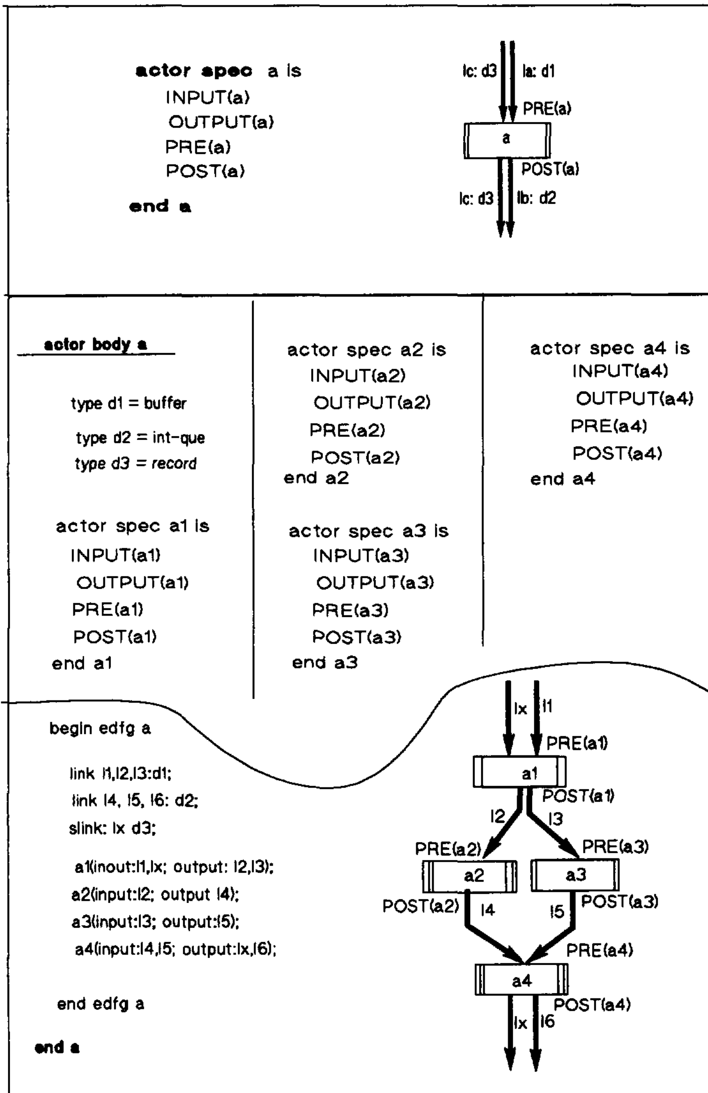


Figure 5.4. An *edfg* and its parent actor.

definition 4 *(syntactic consistency). An edfg is said to be syntactically consistent with an actor, if the external input links of the edfg correspond to the input links of the actor, and the external output links of the edfg correspond*

to the output links of the actor.

According to the definition above, the syntactic consistency of an *edfg* with respect to its parent actor can be mechanically verified according to the signature of the actor and the *edfg*, although the detailed discussion for such a technique is considered trivial in this thesis.

In order to illustrate what we mean by the semantic consistency of a refinement with respect to its parent actor in our approach, we first discuss what the *refinement* is. In some other kinds of design specification, e.g. hierarchical specification, the higher level specification is independent of the lower level specification. This implies that the higher level specification specifies the full requirement that a component should meet. The refinement specifies the details of the component but provides no more decision about the requirement. In contrast, the refinement in a large-scale component may provide complementary information on the requirement. When multiple-level specification is adopted, the requirement may only be partially specified at a higher-level. In particular, the specification of the actor to be refined in terms of an *edfg* may contain only partial information about the properties of the actor, and the additional information can be provided by the specification of the *edfg*.

As indicated by France[40], the semantic consistency of a refinement with respect to an object to be refined can be represented in terms of I/O consistency. The notion I/O consistency captures the black-box approach to refinement, where only the values going into and coming out of an object are of concern. According to such an issue, the semantic consistency of an *edfg* can be defined as follows:

definition 5 *(the connection consistency of actors) Two actors A and B have a connection if an output link of actor A is an input link of actor B. The connection is consistent if the postcondition of actor A concerning the link implies the precondition of actor B concerning the link. i.e.*

$$POST_A(l) \Longrightarrow PRE_B(l)$$

where l is an output link of actor A and an input link of actor B.

definition 6 *(the refinement consistency of an actor) An edfg is said to be consistent semantically with its parent actor if the edfg is syntactically consistent with its parent actor, and the preconditions on the external input-links of the edfg imply the preconditions on the input-links of the actor, and the postconditions on the external output links of the edfg imply the postconditions*

on the output links of the parent actor, i.e.

$$(PRE(edfg) \Longrightarrow PRE(A)) \wedge (POST(edfg) \Longrightarrow POST(A))$$

where PRE(A)/POST(A) are the pre/postconditions on the input/output links of actor A, and PRE(edfg)/POST(edfg) are the pre/postconditions on the external input/output links of edfg.

The definition above is based on the assumption that the actors appearing in an *edfg* are appropriately represented and that the detailed design of the actors is consistent with them. Such an assumption reflects the black-box approach which allows the consistency verification to be done independently at each level of abstraction. This implies the possibility of consistency verification in an early phase of system development, which is often a significant way to reduce the cost of system development and maintenance.

The refinement consistency as defined above refers to the multiple-level specification, i.e. the specification of the parent actor may only provide partial specification of an actor, the further information is provided by the specification of its refinement. For example, assume that A is an actor which has input link l1 and output link l2. A specification of the actor is

PRE(a)=F(l1):	*type(l1)=FILE* $\wedge$ *FILE={Eps-FILE, Txt-FILE}*
POST(a)=L(l2):	*type(l2)=LIST* $\wedge$ *LIST={Int-List, Real-List}*

where $type(x)$ is a function. The input of this function is a link name, and output, the type of the link that the link name refers to. Assume E is an *edfg*, the refinement of actor A, and links l1 and l2 are only external input link and output link of E. The specification concerning the external links of E is

PRE(E)=EF(l1):	*type(l1)=Eps-FILE* $\wedge$ *E-FILE={Std-Eps-FILE, Old-Eps-FILE}*
POST(E)=RL(l2):	*type(l1)=Int-LIST* $\wedge$ *Int-LIST={Pos-List, Neg-List}*

Then, according to the definition of refinement consistency (ignoring the syntax consistency), the following expression must be true.

$$EF(l1) \Longrightarrow F(l1) \wedge RL(l2) \Longrightarrow L(l2)$$

Obviously, a refinement of an actor is only a delegate of all possible refinements of the actor. A refinement inherits all the constraints of its parent actor and has additional constraints. The consistency verification is to verify the consistency of the additional constraints with respect to that of the actor to be refined.

The verification of the semantic consistency of an *edfg*, particularly, a design instance, with respect to its parent actor in a large-scale component includes three steps: (1) Viewing an *edfg* as a set of interrelated boxes, one verifies the consistency between the boxes; (2) Viewing the whole *edfg* as a box, one deals with the consistency of this box with respect to its parent actor; (3) Looking into the boxes — the actors contained in an *edfg* — in order to verify the consistency of the lower-level edfgs with respect to the actors. Step (1) can be done by following definition 5; step (2) can be done by following definition 6; step (3) can be done recursively by repeating step (1) and step (2) until primitive components are reached. Since the use of definition 2 is only concerned with two boxes, the consistency verification is relatively not complex. Our discussion will be centered on step (1), as the interconnections inside an *edfg* often associates several boxes and are relatively complex. The method for dealing with the complexity is *edfg-merging*, which allows the verification in a structured manner.

Our discussion will include two parts. In the first part we discuss the definition of well-formed *edfgs*, as well as their consistency verification. In the second part we discuss the definition and verification for general *edfg-merging*. In each part an example is given.

The consistency verification for well-formed edfgs

definition 7 *(well-formed edfgs). An edfg is well-formed (wf for short) if and only if it is one of the edfgs constructed with constructors (1), (2), (3) and/or (4).*

1. *A single actor with at least one input link and one output link is a wf-edfg. The input (output) links of the actor are the input (output) links of the*

wf-edfg and the pre/postconditions of the actor are the pre/postconditions of the wf-edfg;

2. *Two sequentially connected wf-edfgs form a (sequential) wf-edfg. Two wf-edfgs are sequentially connected if and only if all output links of one wf-edfg are the only input links of the other.*

3. *A splitter is a wf-edfg. A wf-edfg is said to be a splitter if it consists of at least three (sub)wf-edfgs, and all output links of one (sub)wf-edfg constitute all input links of all other (sub)wf-edfgs;*

4. *A binder is a wf-edfg. A wf-edfg is said to be a binder if it consists of at least three (sub)wf-edfgs, and all input links of one wf-edfg constitute all output links of all other (sub)wf-edfgs;*

The constructors 1, 2, 3, and 4 are depicted by (c1), (c2), (c3) and (c4) in figure 5.5 respectively. Although the splitter or the binder contains only three actors as shown in (c3) and (c4) respectively, a splitter or a binder with more actors can easily be extended from them.

Rules for (well-formed) *edfg-merging.* The term *edfg-merging* refers to merging the sub-actors of an *edfg* without loosing its properties for consistency verification. The rules for such a merging are as follows:

1. The rule for a single actor *edfg* is:

$$\frac{\{PRE(a)\}a\{POST(a)\}}{\{PRE(edfg)\}edfg\{POST(edfg)\}}$$

   where $PRE(edfg) = PRE(a) \wedge POST(edfg) = POST(a) \wedge edfg = a$.

2. The rule for a sequential *edfg* is:

$$\frac{\begin{array}{c}\{PRE(edfg_1)\}edfg_1\{POST(edfg_1)\}\\ \{PRE(edfg_2)\}edfg_2\{POST(edfg_2)\}\\ POST(edfg_1) \Longrightarrow PRE(edfg_2)\end{array}}{\{PRE(edfg_1)\}edfg\{POST(edfg_2)\}}$$

3. The rule for a splitter *edfg* is :

$$\frac{\begin{array}{c}\{PRE(edfg_1)\}edfg_1\{POST(edfg_1)\}\\ \{PRE(edfg_2)\}edfg_2\{POST(edfg_2)\}\\ \{PRE(edfg_3)\}edfg_3\{POST(edfg_3)\}\\ POST(edfg_1) \Longrightarrow (PRE(edfg_2) \wedge PRE(edfg_3))\end{array}}{\{PRE(edfg_1)\}edfg\{POST(edfg_2) \wedge POST(edfg_3)\}}$$

4. The rule for a binder *edfg* is :

$$\frac{\begin{array}{c}\{PRE(edfg_1)\}edfg_1\{POST(edfg_1)\}\\ \{PRE(edfg_2)\}edfg_2\{POST(edfg_2)\}\\ \{PRE(edfg_3)\}edfg_3\{POST(edfg_3)\}\\ POST(edfg_1) \wedge POST(edfg_2) \Longrightarrow PRE(edfg_3)\end{array}}{\{PRE(edfg_1) \wedge PRE(edfg_2)\}edfg\{POST(edfg_3)\}}$$

Notice that all *edfgs* in the rules above are well-formed and that the rules for splitter and binder can be similarly addressed in case the actors contained are more than three.

For the well-formed *edfgs*, the process of consistency verification can be viewed as a series of *edfg*-merging steps until the simplest *edfg* is obtained. The simplest *edfg* is an *edfg* which contains one box with explicitly represented *PRE(edfg)/POST(edfg)*. In this case the verification can be completed according to the definition of refinement consistency.

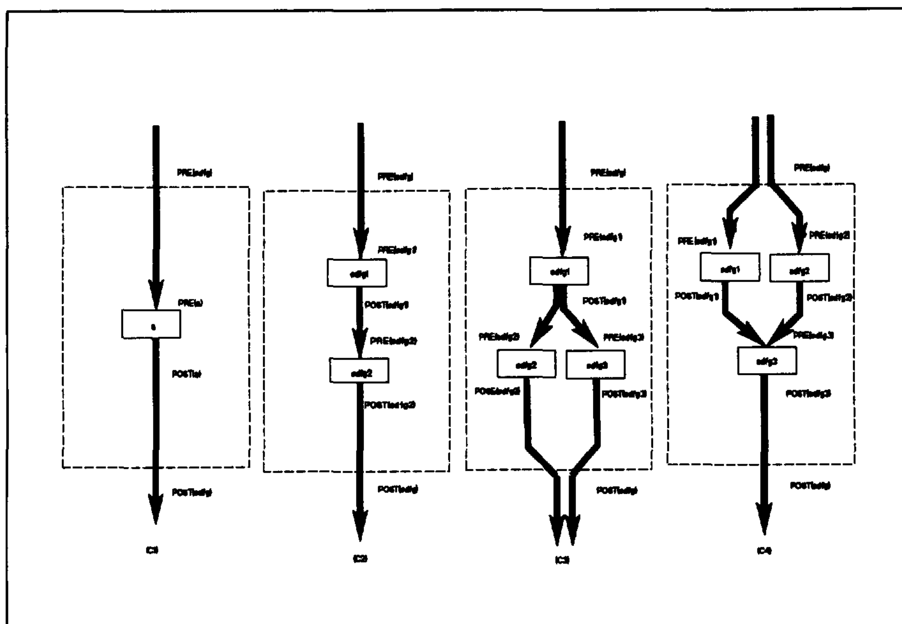


Figure 5.5. The constructors for *wf-edfgs*.

An example of (well-formed) edfg-merging and consistency verification. In figure 5.6 an example is provided for *edfg-merging.* The block (i) is the original edfg. From block (i) to block (ii) rule 1 is applied, which rewrites the original *edfg* into an abstract form.

Since edfgl, edfg2 and edfg3 in block (ii) form a splitter, they are merged into edfgl23 according to rule 3, resulting in block (iii). For such a merging, we need to check

$$POST(a1) \Longrightarrow PRE(a1) \wedge PRE(a2).$$

Similarly, since edfg4, edfg5, and edfg6 in block (iii) form a binder, they are merged into *edfg456*, resulting in block (iv). For such a merging, we need to check

$$POST(a4) \wedge POST(a5) \Longrightarrow PRE(a6)$$

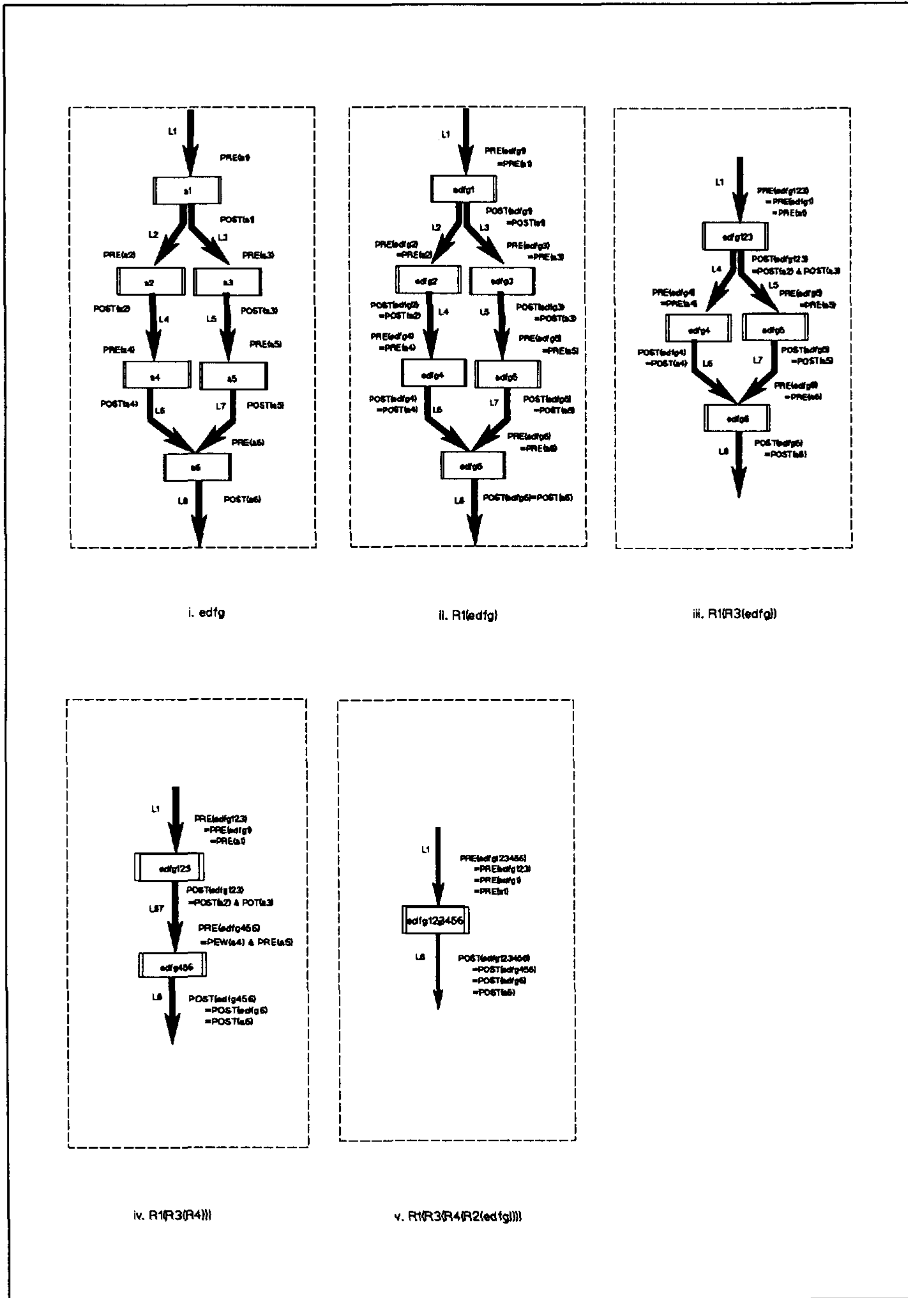


Figure 5.6. An example for (well-formed) *edfg-merging.*

Applying rule 2 to the sequential structure edfg123 and edfg456 in block (iv), edfg123456 is obtained, resulting in block (v). For such a merging, we need to check

$$POST(a2) \wedge POST(a3) \Longrightarrow PRE(a4) \wedge PRE(a5)$$

Finally, the consistency verification can be completed according to the definition of consistency if the following predicate is verified:

$$PRE(a1) \Longrightarrow PRE(A) \wedge POST(a6) \Longrightarrow POST(A)$$

where actor A is the parent actor to be refined.

Finally, we must indicate that — although the notion of well-formed *edfgs* is useful — it forms only a subset of *edfgs* which are defined in chapter 3. In the following part we are going to define general edfgs and to provide rules for their merging.

The consistency verification for well-structured edfg

Well-structured edfgs refer to a broader rang of edfgs than well-formed edfgs. The semantic verification concerned with well-formed edfgs is relatively more complex than that of well-formed edfgs.

definition 8 *(well-structured edfgs). An edfg is well-structured if and only if it can be constructed with constructors (1) and/or (2).*

1. *A single actor with at least one input link and one output link is a well-structured edfg. The input (output) links of the actor are the input (output) links of the edfg and the pre/postconditions of the actor are the pre/postconditions of the edfg;*

2. *Two well-structured edfgs A and B form a well-structured edfg if a non-empty external output-link-set of A goes to B directly and all other output links of A do not go to B directly or indirectly.*

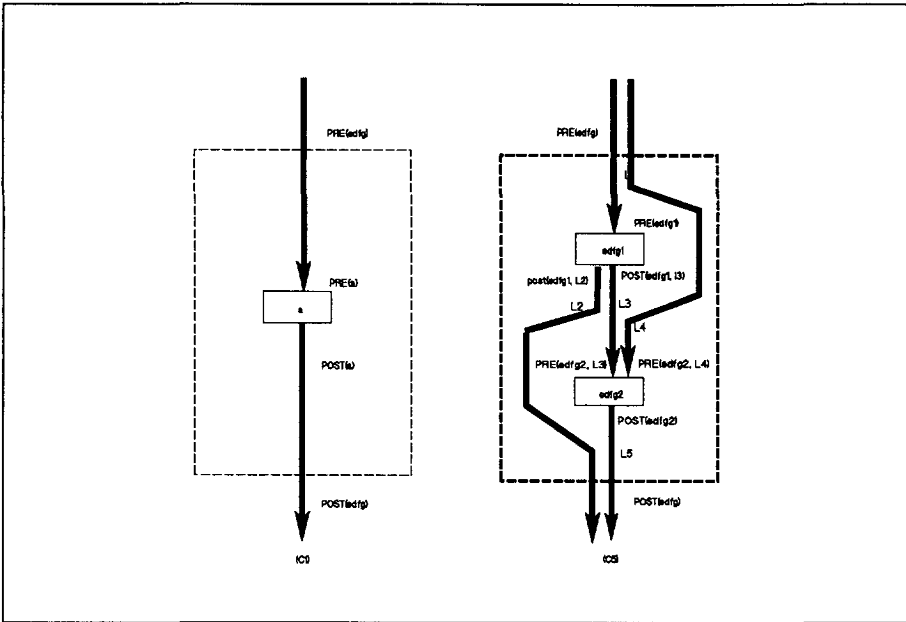


Figure 5.7. *The constructors for well-structured edfgs.*

Notice that the link set L2 and L4 can be empty, while the link set L3 must be non-empty. Obviously, the notion of well-structured edfgs is more general than the well-formed edfgs, as all well-formed *edfg* constructors can be constructed in terms of well-structured *edfg* constructors.

Rules for well-structured ***edfg-merging*****.** The rules for well-structured *edfg-merging* are given corresponding to the rules for the construction of such edfgs.

1. The rule for a single actor *edfg*

$$\frac{\{\mathrm{PRE(a)}\}\ \mathrm{a}\ \{\mathrm{POST(a)}\}}{\{PRE(edfg)\}edfg\{POST(edfg)\}}$$

where

$$PRE(edfg) = Pre(a) \wedge POST(edfg) = POST(a) \wedge edfg = a.$$

2. The rule for well-structured *edfg* constructor

$$\frac{\begin{array}{c}\{PRE(edfg_1)\}edfg_1\{POST(edfg_1, L2) \wedge POST(edfg_1, L3)\} \\ \{PRE(edfg_2, L3) \wedge PRE(edfg_2, L4)\}edfg_2\{POST(edfg_2)\} \\ POST(edfg_1, L3) \Longrightarrow PRE(edfg_2, L3)\end{array}}{\{PRE(edfg_1) \wedge PRE(edfg_2, L4)\}edfg\{POST(edfg_1, L2) \wedge POST(edfg_2)\}}$$

where

$$L2 \cap L3 = L3 \cap L4 = \{\}$$

The expression *PRE(E,L)/POST(E,R)* refers to the preconditions on input link set L before executing the *edfg* E and to the postconditions on output link set R after executing the edfg.

An example for (well-structured) ***edfg-merging*** **and consistency verification.** The process of merging a well-structured *edfg* is shown in figure 5.8. The first step, the step from block (i) to block (ii), is the same to that of merging well-formed edfgs.

Since edfg1 and edfg2 in block (ii) form a constructor, they are merged into edfg12 according to rule 2, resulting in block (iii). For such a merging, we need to check

$$POST(a1, L3) \Longrightarrow PRE(a2, L3)$$

Since edfg12 and edfg3 in block (iii) form a constructor, they are merged into edfg123 according to rule 2, resulting in block (iv). For such an *edfg-merging*, we need to check

$$POST(a1, L4) \wedge POST(a2, L6) \Longrightarrow PRE(a3, L4) \wedge PRE(A3, L6)$$

Since edfg123 and edfg4 in block (iv) form a constructor, they are merged into edfg1234, resulting in block (v). For such an *edfg-merging*, we need to check

$$POST(a2, L5) \Longrightarrow PRE(a4)$$

Finally, according to definition 6 verification can be completed by verifying the following predicate.

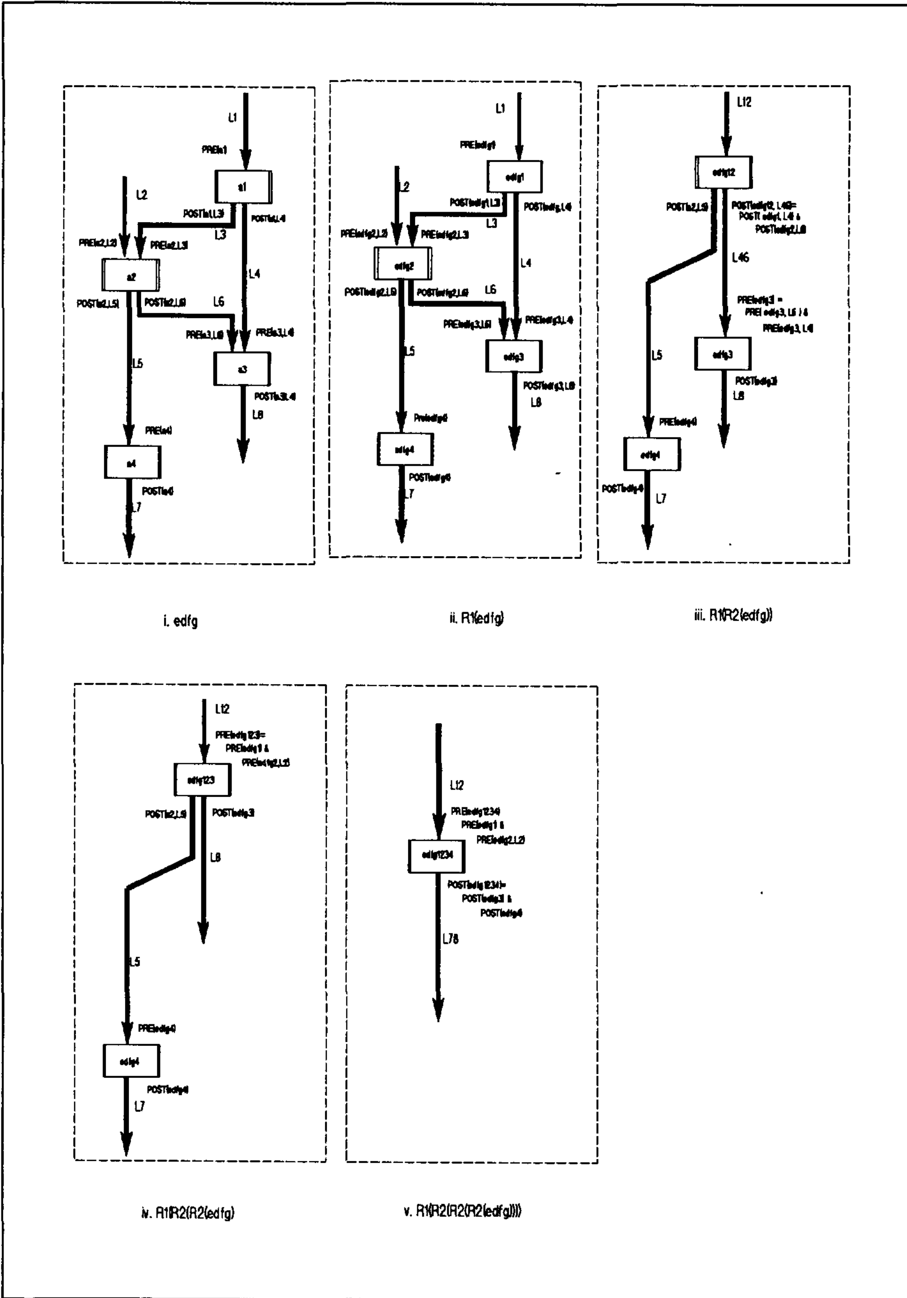


Figure 5.8. An example for (well-structured) *edfg-merging.*

$$((PRE(a1) \wedge PRE(a2, L2)) \Longrightarrow PRE(A)) \wedge$$
$$((POST(a3) \wedge POST(a4)) \Longrightarrow POST(A))$$

where actor A is the parent actor of the edfg.

Additionally, notice that block (iii) is already a well-formed edfg, it is possible to merge this block to block (v) directly.

5.4 Summary

In this chapter we discussed a principle for design specification, we described how to apply such a principle for the representation of large-scale components, and we introduced a method for consistency verification.

Particularly, we discussed the idea of multiple-level specification for the specification of large-scale components, and described how to specify design frameworks, domain models, design instances and levels of refinements. The multiple-level specification was reflected in the representation of large-scale components in two dimensions, i.e., vertically, in the relationships between different levels of refinement, and horizontally, in the relationships between domain model, design framework and design instance. The advantages of using multiple-level specification is the capability of specifying large-scale components without loosing their reusability.

As an effort to stress both the application of formal methods and the advantages of using data flow graphs, several papers have recently been published as a combination of data flow graphs and formal specifications such as [27][42] [105], emphasizing building a connection of graphic notation with formal semantic specification. More recent work is Larsen's method and France's approach. The former emphasizes a transformation from a kind of data flow diagram into a VDM specification, the latter specifies in the semantics of diagram notation directly. Our method suggests a diagram guided semantic verification, by which a refinement verification is straightforward and efficient. It is straightforward as the verification can be done with semantically extended data flow graphs. It is efficient as such a verification can be simplified with a set of inference rules and guided with graphical structures which can be intuitively understood.

Chapter 6

Reuse-in-the-Large and Application Prototyping

Prototyping originated from those engineering disciplines which are involved in mass production.

—Sharam Hekmatpour & Darrel Ince, 1988.

In this chapter the model for reuse-in-the-large is applied to application prototyping with a combination of this model and the EDFG approach.

6.1 Introduction

Application prototyping refers to the practice of building an early version of an application system which does not necessarily reflect all features of the final system, but rather those which are of interest[48]. The methods applied to application prototyping include *throw-it-away prototyping*, *incremental prototyping* and *evolutionary prototyping*. We prefer evolutionary prototyping, since such a method allows the inconsistencies and shortcomings of an application to be experimentally uncovered, and the application developer and customer to experimentally learn more about their own needs [48]. According to this method, the process of application development is to continually generate a series of experimental versions of an application and each new version of the application is built by the modification and reconstruction of existing versions. Such a process certainly implies a kind of application reuse. The question is

what methodology is a good choice to support such kind of reuse. In this chapter we argue that applying RITL to such kind of reuse will lead to rapid application prototyping.

As discussed before, RITL suggests an integrated representation of both the large-grain code components and the design information. Such a representation is called a large-scale component, providing a basis to generate a new version of application by the reuse of design information and implementation of existing applications and components. To set an example, we discuss how this method may support the EDFG approach.

In section 6.2 the concrete representation of large-scale components is discussed in terms of extended data flow graphs (EDFG). In section 6.3 we discuss the process of application prototyping in terms of RITL.

6.2 Representing large-scale components in terms of EDFG

In this section we discuss prototypes and large-scale components in terms of EDFG concepts, which may, as discussed later on, lead to a particular method for application prototyping concerned with RITL. Firstly, the concepts about EDFG are introduced, then prototypes are defined in terms of EDFG and, finally, a large-scale component is represented in terms of prototypes.

6.2.1 The concepts of EDFG

Most EDFG concepts were introduced in chapters 3 and 5. Now, we will provide some more concepts which are capable of forming a basis to demonstrate our method of applying RITL to application prototyping in this chapter.

definition 9 (loosely constrained EDFG and actor). *An EDFG is said to be loosely constrained if there is at least one loosely constrained actor in the EDFG. An actor is said to be loosely constrained if it is a loosely constrained EDFG or it may be refined or implemented with different components alternatively (genericity).*

The components refer to either the components of an application design or the components of an application implementation such as EDFGs, actors, function, data abstraction and so forth.

definition 10 (stronger constraints). *The constraints on an EDFG E1 (actor a1) are said to be stronger than the constraints on an EDFG E2 (actor a2), iff the corresponding specifications or implementations of E2 (a2) satisfy the specification of E1 (a1) (specificity).*

The constraints in definition 10 refer to logical assertions described in[47].

definition 11 (the refinement of an EDFG) . *The refinement of an EDFG is also an EDFG which inherits all the properties of the refined EDFG and replaces at least one actor of the refined EDFG with a sub-EDFG. The sub-EDFG satisfies the specification of the replaced actor.*

6.2.2 The process of EDFG decomposition

According to the EDFG approach, the process of application design starts from the creation of a highly abstract EDFG and follows a series of refinements until the actors contained in the refined EDFGs are found in a component base, or the actors are small and simple enough to be directly implemented easily. Refining an EDFG is replacing at least one actor of the refined EDFG with a sub-EDFG. The implementation of a designed application is the implementation of all actors of the finally refined EDFGs[65].

In order to understand the refinement of an EDFG, it is necessary to understand the relationship between an EDFG and its refinement. An EDFG to be refined is an abstraction which captures the commonality of a set of alternative refinements. For example, when we assume that an EDFG contains an actor 'sort', the actor may be refined by either sub-EDFG *quick sort, bubble sort, hash-sorting or heap sort* and so on during a refinement. During application prototyping the decision on selecting an instance (sub-EDFG) from the set of choices is allowed to be made experimentally.

As an experiment, like the experiment in chemistry and physics, the prototyper would be very much concerned with the subsequent choice. Therefore, we need a mechanism to record the process of the prototyping step by step from the highest abstraction of EDFG to the final implementation. In order to support an experimental design, for each abstraction in a step, three pieces of information are needed: (1) a generic EDFG which presents the common structure of a set of alternative designs at an abstraction level; (2) resources to instantiate the generic EDFG into alternative designs; (3) a particular EDFG

which presents the currently selected design. The three pieces of information may form a *prototype*, as described later on, and a hierarchical structure consisting of a set of prototypes at different levels of abstraction will form a large-scale component, which can be a basis to produce a variety of applications in an application domain.

6.2.3 The definition of prototypes in terms of EDFGs

In order to represent a concrete large-scale component in terms of EDFG, we first define a *prototype*. A prototype as shown in some delegate-based languages, is defined as an object which serves as both an instance and a template[126]. The advantages of such a definition are, that the instance can present the default for the operations and the value of the prototype, and the template can be a basis to generate a set of alternative instances, which could be convenient for prototyping. However, the information contained in the prototype is limited only by the constructs of the programming language. In order to support RITL, we further extend the definition of a prototype to represent a design including design instance, design framework and domain resources.

definition 12 *A prototype is defined in terms of EDFG as follows.*

$$prototype=(edfg, \underline{EDFG}, domain)$$

- *The $\underline{EDFG}$[1] is a design framework, which represents a loosely constrained EDFG or a loosely constrained actor. An EDFG captures the commonality of a set of edfgs, which can be instantiated in terms of domain resources, in order to produce alternative edfgs.*
- *The domain is short for domain resources, which includes a set of specifications and a set of entities. The specifications express what kind of components may be used to form alternative edfgs, i.e. the instances of $\underline{EDFG}$. The entities may be large-scale components or the implementation of actors or data abstractions.*

[1]Note that in the following part of this chapter we use underlined "$\underline{\text{EDFG}}$" to stand for the template of a prototype and small case italic "*edfg*' for the instance of a prototype.

- *The edfg is a particular design instance, a default instance of <u>EDFG</u>. The constraints on the edfg are stronger than (or equal to) the constraints on the <u>EDFG</u>. The functionality of the edfg delegates the functionality of this prototype.*

According to the definition above, we could create a prototype with a 3-tuple as in figure 6.1. The actors A_0, A_{01} and A_{011} are loosely constrained; and the type F and T are union types (see also 4.4.1), F=[..., file,...] and T=[...,tree, ...]. The *edfg* describes a more peculiar component than the <u>EDFG</u>. The constraints on the actors a_0, a_{01} and a_{011} of the *edfg* are stronger than the constraints on the corresponding actors A_0, A_{01} and A_{011} in the <u>EDFG</u>.

The *domain* defines a set of components which may be used to instantiate the <u>EDFG</u>. For the textual representation of *edfg*, <u>EDFG</u> and domain specification, see also 3.2.4 and 3.4.1. For the repository of the prototype, see also 4.4 and 6.3.3.

There is another example of a prototype in figure 6.2, which contains quite some constraints on the <u>EDFG</u> of a prototype. The links l_1, l_2, l_3, l_4, l_5,and l_6 are constrained with data types and the link type U is a union type so that U=[..., tree, ...]. The domain specification in this example is limited to two actors of the <u>EDFG</u>.

6.2.4 Representing large-scale components in terms of prototypes

With prototypes as defined before, a large-scale component can be described by a hierarchical structure, every node of the structure is a prototype, see also figure 6.3. The *edfgs* of the large-scale component at one level are the specifications of the *edfgs* at a lower level of the hierarchical structure. For example, in the large-scale component-A, the *edfg* of a_1 is the specification of the *edfgs* a_{11},a_{12} and a_{13} together, see figure 6.3 and the left part of figure 6.4, and the *EDFG* of a_{12} is the specification of the *edfgs* of a_{121}, a_{122} and a_{123} together, see figure 6.3 and the right part of figure 6.4. Apart from being easy to understand, a significant characteristic of the large-scale component is its large-grain and genericity. For example, based on large-scale component-A, various different large-grain components, i.e. peculiar designs and implementations, can be obtained by adjusting the *edfg*s of the prototypes in the

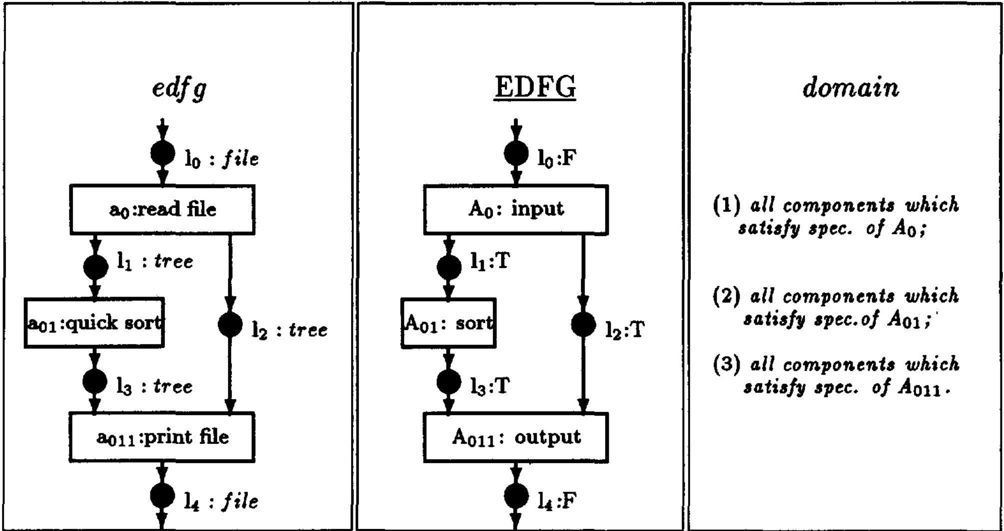


Figure 6.1. An example of prototypes.

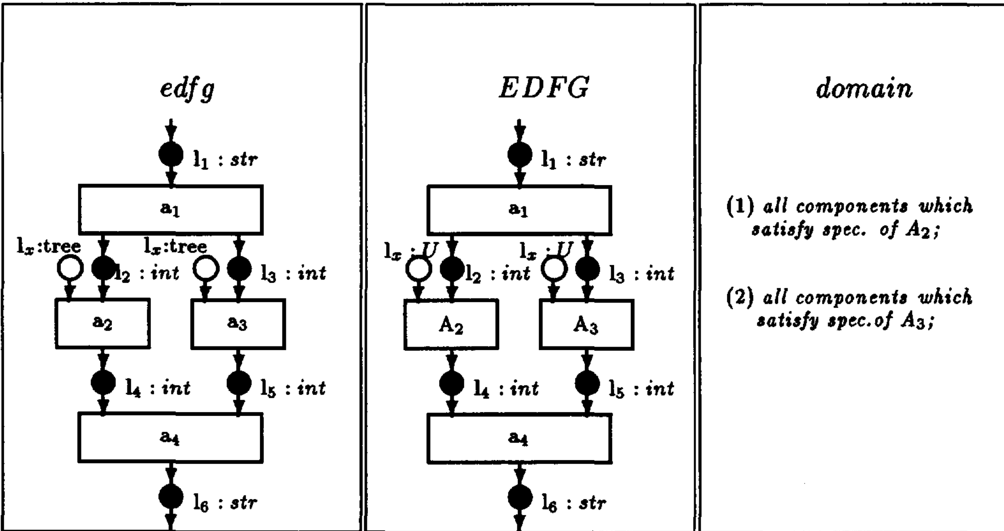


Figure 6.2. An example of constrained prototypes.

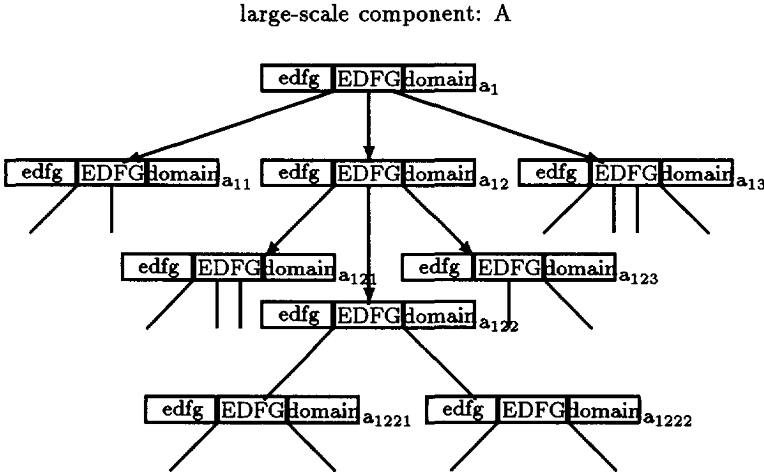


Figure 6.3. An example of a large-scale component.

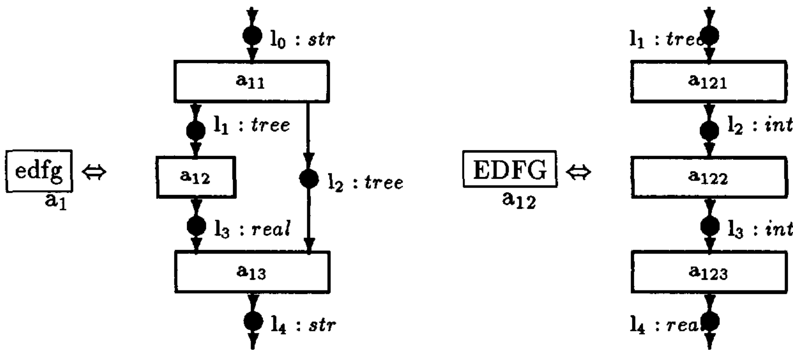


Figure 6.4. The refinement of an *edfg*

large-scale component. The adjustment of the *edfgs* is involved in applying the *EDFG mapping* to different elements of its domain and therefore producing a different *edfg*. The EDFG mapping will be discussed later on. The large-grain component may be different for any change of any *edfg* of the component.

6.3 Using large-scale components in application prototyping

In this section we discuss the method of applying large-scale components to the process of application prototyping.

6.3.1 The process of application prototyping

The genericity of a large-scale component makes the large-scale component itself reusable, especially in application prototyping when many different instances of a large-scale component may be required for experimenting a target application system. The process of application prototyping with large-scale components is described in figure 6.5. The process starts from *building* a large-scale component, then goes through a loop *execution, evaluation and modification* until the *target application system* is obtained.

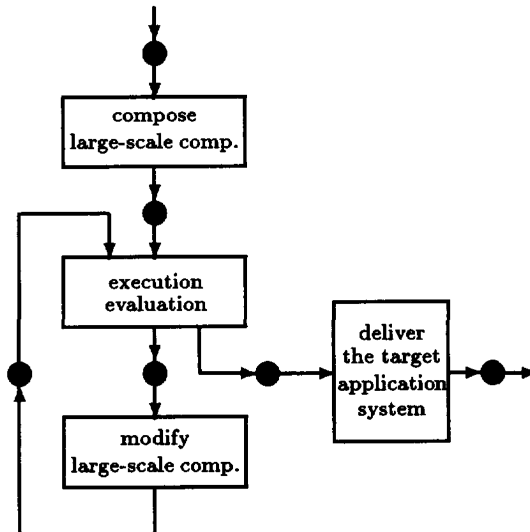


Figure 6.5. The process of application prototyping.

Building large-scale components

The process of building a large-scale component follows the process of top-down decomposition. This can be started by creating a prototype which represents the application at the highest level of abstraction, and then refining the prototype with a set of lower level prototypes until the prototypes to be refined become primitive ones. A primitive prototype is a prototype, the *edfg* of which contains only primitive actors and these actors are implemented in terms of code components.

Refining prototypes is to refine the *edfg* of a prototype, as shown in figures 6.3 and 6.4. A new EDFG which represents the (generic) decomposition of an actor of the refined *edfg* is the EDFG of a new prototype. The new *edfg* of the new prototype may be generated by proposing more constraints on the EDFG. Then the new prototype is made by adding the *domain* according to the requirement of the EDFG, including both the domain specification and the domain entities.

Executing large-scale components

It is known that the most significant characteristic of application prototyping is to allow designers and users to exchange their ideas about the target application system over an executable (experimental) system. This can be supported by large-scale components. A large-scale component can be used to present an executable subsystem or system if all lowest level refinements of the large-scale component end with primitive prototypes.

Modifying large-scale components

After executing the experimental system, designers should make decisions about redesign or modification of the experimental system. The redesign or the modification often refers to the reconstruction of the design instances (*edfg*s here) which are contained in the prototypes of a large-scale component. Such a redesign or modification can be guided by browsing the structure of the large-scale component hierarchically. Once a prototype (a node of the hierarchical structure) is located for a redesign or modification, there are two possibilities to do so. First, one may make a new *edfg* by EDFG *mapping*, i.e. by the instantiation of the EDFG of this prototype. Because of the *domain* of the

prototype, there are a set of alternative entities which can be used for the instantiation. Secondly, one may want to rebuild the lower level design of a particular actor of the current *edfg*. This can be done by looking into the refinement of the actor, i.e. looking into the corresponding lower level prototype. Inside the prototype at the lower level, a new *edfg* can be produced by the EDFG mapping at this level. Since the EDFG at this level is the (generic) decomposition of the actor at the higher level as discussed before, the new *edfg* can meet the requirement of the actor at the higher level. The consistency of an *edfg* with the requirement described by the specification of the higher level actor can be verified semantically, also recall chapter 5.

Sometimes, the resources contained in the domain may not be sufficient to build a new *edfg*, then new resources must be supplied by reusers. If the supplied resources satisfy the domain specification, they can be added to the domain resources for reuse later on.

Delivering the target application system

The target application system of prototyping is the result of experiments with a series of experimental systems. The relationship between the large-scale component-A and the prototyped subsystem-A is shown in figure 6.6. As the results of prototyping, we may not only have a specific application system, but also a set of prototypes, a large-scale component and a set of sub-large-scale components. As these objects have genericity, they are usually regarded as resources to be reused later on. More important, the large-scale component as a basis to generate the delivered application system can be further used as a basis for the maintenance of this system. In this case the system maintenance can be regarded as the continuation of the prototyping. This may further reduce the cost of system maintenance, because the large-scale component is easy to be understood and modified.

Consequently, application prototyping with large-scale components provides not only possibilities to reuse system components (code components and design information), but also possibilities to create the artifacts for reuse, by which the reusability of a large-scale component can be increased and domain analysis can be incrementally completed.

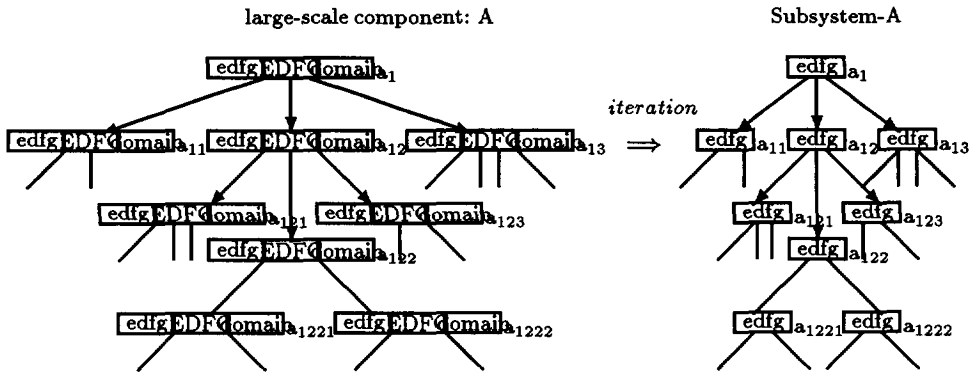


Figure 6.6. A large-scale component and the prototyped application system.

6.3.2 Component specification and EDFG mapping

It is known that application reuse can be significantly supported in terms of specification and the manipulation of system components. In this subsection we discuss how to specify and manipulate large-scale components for application prototyping. The component manipulation refers to the EDFG mapping here.

Specifying prototypes

Since a large-scale component can be viewed as an organization of a set of prototypes, a language is required for the specification of the prototypes and the relationships between them.

The language to specify prototypes should be able to specify *edfg*, <u>EDFG</u> and domain. The specification of an *edfg* includes graphical specification and textual specification. The former demonstrates the structure of a component and the latter provides the interface and the semantics of the component precisely. Both of them are helpful to the practice of application development. Our method is to provide a textual specification which can be transformed

into a data flow graph[74]. There are two cases: (1) if an *edfg* is an actor, it can be specified by an actor *module* as in figure 3.2; (2) if an *edfg* is an EDFG, it can be specified by a *package* as represented in the left part of figure 3.6. The module shown in figure 3.2 can directly be derived from the definition of an actor. The package shown in figure 3.6 specifies an *edfg* in terms of link declarations and the signatures of the actors.

In a package the link declarations declare the data types which the tokens of the links and state links (slink) may have. The actor declarations declare the names of actors and the *input* links and *output* links of the actors as well. Since different actors may share the same link, the connection between actors is reflected in the package. Such a package can be transformed into a data flow graph as shown on the right of figure 3.6. The semantic specification of the *edfg*, and of each actor contained in the package, can be additionally provided as shown in figure 3.5.

The specification of an EDFG is similar to the specification of *edfg*. The only difference is that the EDFG is a more loosely constrained EDFG than the corresponding *edfg*. The difference can be reflected by allowing the module or the package to have generic parameters as discussed in section 3.4. A generic parameter is a parameter of components, the values of which may be modules, including source code or large-scale components.

If the genericity of the EDFG is specified in terms of generic parameters, the domain specification may specify the domain of the generic parameters. This can be done in terms of actor modules and the notions in some specification languages such as VDM[54], Z [60] or RSL[47].

EDFG mapping

EDFG mapping refers to the activity of instantiating an EDFG with domain resources into an instance of the EDFG. The instance may be directly an *edfg* or further specialized into an *edfg*. The process of an EDFG mapping is:

1. retrieve existing components (the actual parameters of the EDFG) which satisfy the domain specification;

2. select proper components from the retrieved components, if any, or create proper components satisfying the domain specification;

3. transform the selected components and the EDFG into an instance of the EDFG.

4. specialize the instance of the EDFG if necessary.

As an EDFG may be expressed by a parameterized module (for actor) or a package (for EDFG), the process of transforming the EDFG and the components of the domain into an *edfg* is equivalent to the process of binding the parameters of an EDFG with the components. This is largely similar to instantiating a generic component in Ada and can, therefore, be conducted automatically. However, this does not imply that the EDFG mapping is a generic instantiation as in Ada. The semantics of the generic parameters of an EDFG is completely provided by domain specification, whereas the semantics of the generic parameters of generic components as in Ada is not sufficiently described[80]. Furthermore, the domain and the range of the EDFG mapping may not only include source code but also, more important, large-scale components containing design instance, design framework and design resources.

Additionally, for a specialization, an EDFG mapping may not only include the instantiation of the EDFG, but also the activity to add additional actors or links to the instance of the EDFG, resulting in an *edfg*.

6.3.3 The management of large-scale components

In this subsection we discuss the possibility of automatic retrieval and the requirement of a component base for the management of large-scale components.

The possibility of automatic retrieval

It is known that efficient retrieval of components to meet the requirement of an application is of key importance for component reuse. However, the techniques for such a retrieval are currently not efficient, i.e. the cost for the retrieval is still very high. A special issue on this topic concerned with EDFG mapping is discussed in this paragraph.

The major activity of an EDFG mapping is to generate an instance of the EDFG. The basic condition to realize the mapping is to provide necessary components as actual parameters of the EDFG. Instead of creating those components required from a rough draft, we will try to retrieve them from the component base. The retrieval may be done by specifying the requirements in terms of a query language of a component base. However, such a specification implies transforming the domain specification into the specification for entity retrieval. The domain specification may be described in terms of a specification

language; the specification for retrieval may be described in terms of a query language of the data base. As the two languages are usually very different from each other, the transformation is often complex if not impossible.

In order to reduce the cost for the transformation and, therefore, to speed up EDFG mapping, we propose the reuse of the retrieval specification; i.e. connecting to the domain of each prototype there may be an existing retrieval specification which can be used to retrieve the components of the domain directly. Since the domain specification of a prototype is normally unchanged, such a retrieval specification may be provided by the prototype producer or by the reuser when there is a successful retrieval. The specification is reusable if the connected prototype is reusable. Therefore, we have reason to let every prototype be prepared for reuse to be connected to a retrieval specification.

The storage of retrieval specification practically classifies the components of the component base according to the domain specification. Such a classification is dynamically done by either designer or reuser. The result is automatic retrieval.

The possibility of avoiding retrieval

Automatic retrieval is the reuse of the information for component retrieval, i.e. a reuser may not need to retrieve if the similar retrieval has been done by some earlier retriever. However, the retrieval problem has not been solved yet and retrieval is still necessary. Instead of trying to solve the well-known retrieval problem directly, we tried to find a way to avoid the problem. By observing the process of application prototyping, we find that the reuse activity for a system component often happens at the place where this component is created. For example, when we decompose an operating system at top-level into subsystems *process management, file management, device management, job management* and *memory management*, we need to create these subsystems. In many cases the reuse of these subsystems for the prototyping of varieties of operating systems happens at the same place. Therefore, the retrieval activity can be avoided if we put the component to be reused at the place where it is created. Moreover, as discussed before, the H-structure of a large-scale component provides the data structure for the repository of the created components.

Consequently, either the automatic retrieval or the way of avoiding retrieval provides partial solution of retrieval problem. A component base may support both of them.

The component base

In order to support application prototyping concerned with RITL, a component base is required for repository, classification and retrieval of components.

The general requirement of the repository and retrieval may be realized by a data base[88], and the general component classification can be done, based on the method given by Prieto-Diaz [100]. Particular requirements are that the component base should be capable of dealing with both the structures of components and the relationships among components. This is because large-scale components are structured components and EDFG mapping is concerned with the relationships among a set of components. Such requirements can be met by developing the component base based on a semantic data base management system[11], or an object-oriented data base management system[2]. Those data base management systems provide means to conveniently describe and deal with both the structures of objects and the relationships among the objects. The component base concerning our research, supporting rapid prototyping, is a semantic data base as discussed in chapter 6.

6.4 Summary

In this chapter we discussed how RITL is applied to evolutionary application prototyping. First, we described how large-scale components can be represented in terms of the EDFG notation; then we discussed how to apply large-scale components in the process of application prototyping. We conclude that RITL is capable of building a basis for prototyping varieties of target application systems in an application domain. In terms of RITL, the process of application prototyping can be effectively improved. This is because the information to be reused for prototyping includes different levels of design representations, such as design instance, design structure, and domain resources. Moreover, the information to be reused is often at the hands of reusers once the reuse is necessary. For example, when a designer is going to modify *quick-sort*, i.e. a design instance of a node in a large-scale component as in figures 6.3 and 6.1, a group of alternative components for sorting can be picked up from the domain resources of the same node.

Chapter 7

Reuse-in-the-large: A case study

Information systems are designed and implemented in a rapidly changing and turbulent environment.

— Henk G. Sol, 1991

In this chapter we discuss how reuse-in-the-large is realized within an envisaged support environment, regarding the *Student Management System* (SMS). Particularly, we discuss how the credit-reporter, a subsystem of the SMS, is specified in terms of an available specification language, how it is understood and manipulated and how varieties of designs and implementations can be generated from the large-scale component, so as to meet varieties of requirements in an application domain.

7.1 Introduction

When an approach is addressed and developed to a certain degree, a prototype of such an approach is often useful to convince users with an intuitive and close view of the approach and with practical experience concerning the application of the approach. We will provide a prototype for reuse-in-the-large in terms of large-scale components in an application domain, regarding the practical specification of a large-scale component, the user interface of the large-scale component, the generation of alternative designs and implementations from

the large-scale component, and the practical method for incremental domain analysis.

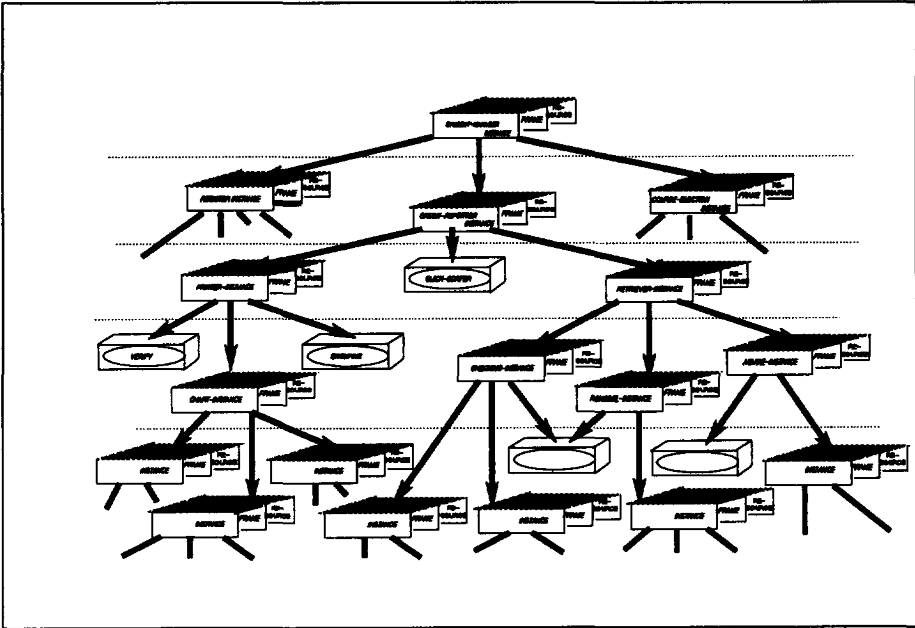


Figure 7.1. The H-structure of the sms

We assume that an envisaged support environment for the prototype includes a semantic database for the management of the large-scale component and a tool for browsing, creation and modification of the large-scale component as described in chapter 4. Moreover, it also includes a formal specification language for component specification.

The applications in this case study are student management systems (SMS). An SMS is defined preliminarily as an application system for student registration, course-election and credit report. We assume that different universities or even different departments may have different requirements or regulations for student registration, course-election and credit report. We also assume that — after a certain lapse of time — the requirements or regulations for a single university or even a single department will be changed. We assume that a set of alternative SMSs are needed practically, so that an application domain exists.

According to the method for reuse-in-the-large in this dissertation, the design information and the source code of the SMS can be organized into a hierarchical structure, representing a large-scale component. The hierarchical structure of the SMS is partially depicted in figure 7.1, the top-node of the hierarchical structure is sketched in figure 7.2, and some outputs of the SMS are shown in figure 7.3.

According to the preliminary decomposition as shown in figure 7.2, an SMS consists of the subsystems: *student-register, credit-reporter and course-election.* The student-register is used for student admission and course registration, the credit-reporter is used for analyzing the progress of the students, and the course-election serves for arranging lectures, laboratories and class rooms, and providing the time schedules for both teachers and students.

Instead of describing the whole system broadly in a chapter, our discussion will be focused on the credit-reporter. With such an example we discuss how a large-scale component is specified in terms of a specification language, how it is understood and manipulated, and how varieties of design instance and implementations can be generated from the large-scale component.

We select the *credit-reporter* also as an example because such a component is an instance of the *data-processor* which has been discussed before. The reasons for such a decision are (1) readers may directly catch the points to be described instead of bothering with additional information about a completely new application, (2) the credit-reporter is complex enough to demonstrate the capability of reuse-in-the-large, and (3) it allows us to represent the language independency of representing large-scale components by specifying the same large-scale component in a different language.

As discussed before, a large-scale component is first created based on the development of a particular component. The information contained in the large-scale component can be incrementally accumulated towards a complete domain analysis. However, in order to be brief and to the point for illustrating how reuse-in-the-large happens regarding a large-scale component, we assume that C-REPORTER, a large-scale component, has already been applied to develop several similar but different credit-reporters rather than at the state of initialization. For the same reason we will try to ignore any trivial descriptions about the component and its support environment, although these are necessary in building a real system. The C-REPORTER is an envisaged

large-scale component, which contains the information accumulated from the development of several alternative *credit-reporters*, or *c-reporters* for short. A c-reporter can be as simple as a small program concerning simple input, sorting and output of credit lists, or as complex as a subsystem concerning data description, data retrieval, analysis, classification, graphic simulation and so on. Nevertheless, all of them are integrated into the large-scale component.

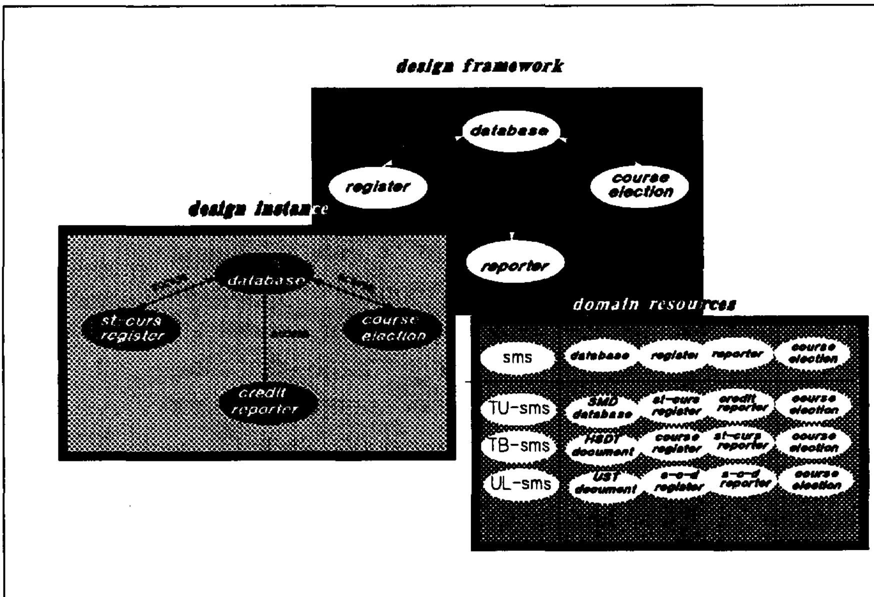


Figure 7.2. The top-node of the SMS.

In section 2 we describe the specification of a particular large-scale component. In section 3 we illustrate how a large-scale component can be browsed and manipulated in terms of a user interface. In section 4 we discuss how the designs of alternative large-grain components can be generated from an existing large-scale component. In section 5 we describe how the implementation of alternative large-scale components can be generated from the existing large-scale component.

Additionally, in order to understand the example precisely, we must remember the distinction between large-scale components (see also section 4.2) and large-grain components (see also section 1.1.3). The former refers to the design information about developing a set of products in an application do-

main, whereas the latter refers to a particular product, although a large-scale component always contains a large-grain component as discussed before.

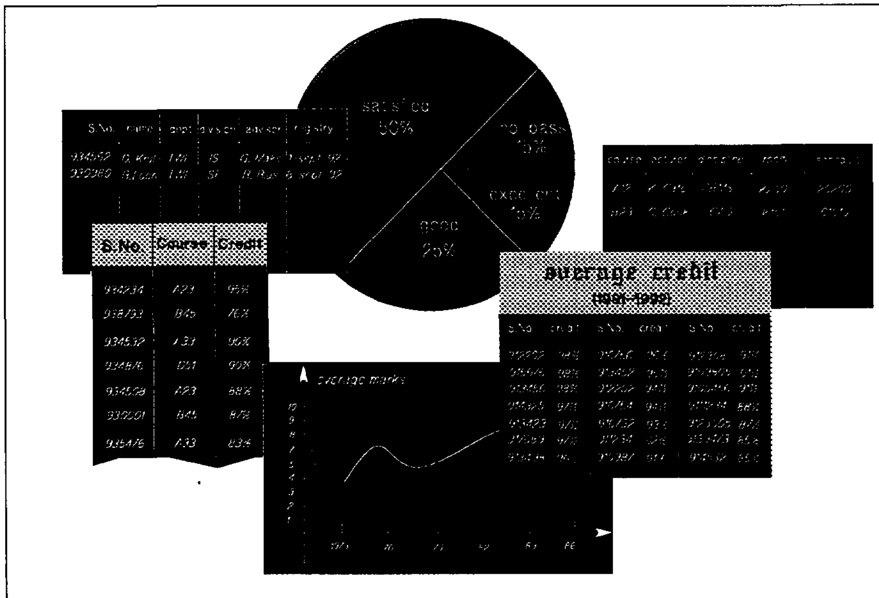


Figure 7.3. Some outputs of the SMS.

7.2 Representing a large-scale component

In this section we provide the representation of the C-REPORTER, a large-scale component. Its H-structure is sketched in figure 7.1; its specification can be given in RSL [1] and diagram notation.

The specification of the first several nodes of the large-scale component are the C-REPORTER, the PRINTER , and the RETRIEVER. Each of them consists of three pieces of description: a MODEL (domain model), a FRAME (design framework) and an INSTANCE (design instance). The INSTANCE further consists of INSTANCE-ENTITIES (the subcomponents in the design instance) and INSTANCE-SUBSYSTEM (the structure of the design instance). Moreover, corresponding to each design FRAME or INSTANCE, there is a diagram by which the organizational structure is specified intuitively.

[1]RSL is a trademark of Computer Researches International A/S.

The specification is a multiple-level one. The subcomponents (entities) appearing in a design instance can be classified into primitive subcomponents and non-primitive subcomponents. The specification of primitive subcomponents must be fully provided, whereas the specification of nonprimitive subcomponents might be partially provided and needs to be supplemented latter on.

Additionally, we would like to indicate that the name C-REPORTER is not only the name of the large-scale component it stands for, but also the name of the top-node of the large-scale component. It is only a convention to use the name of the top-node as the name of the whole large-scale component.

In the remaining parts of this section, we will discuss the specifications, regarding the C-REPORTER (the top-node), the PRINTER (one of the two second-level nodes), and the RETRIEVER (the other second-level node) respectively.

7.2.1 Specifying the C-REPORTER (top node)

In this subsection the top-node of the large-scale component is specified in four text files: document 1.1, document 1.2, document 1.3a and document 1.3b. Document 1.1 contains the domain model of the C-REPORTER; document 1.2 contains the design framework of the C-REPORTER; document 1.3a and 1.3b contain the design instance of the C-REPORTER, including the subcomponents the design instance consists of and the structure the design instance is composed with.

The Domain model of the C-REPORTER

The domain model of the C-REPORTER is specified in terms of the C-REPORTER-MODEL, a scheme in document 1.1.

A scheme in RSL[47] is either a class or a parameterized class. A class is a collection of models. A parameterized class is a mapping from lists of objects to classes: each object list is mapped to a class. An object is either a model or an array of models. A model in RSL is an association of names with entities: each name is associated with a single entity.

The attributes of the domain model of the C-REPORTER (top-node) are specified in this scheme, including *producer, sorter* and *deliver*. The *producer* is a function space from type *Desc* to type *List* $\times$ *Message*; the *sorter* from

List to *List*, and the *deliver* from *List* $\times$ *Message* to *File*.

In the axiom part of the scheme some quite general properties of *producer, sorter* and *deliver* are specified. More specific properties of them will be supplemented in design instance, if necessary. In other words, the general properties specified here will be inherited by the design instances, which will be discussed later on.

The design-framework of the C-REPORTER

The design framework of the C-REPORTER is defined as an extension of the domain model. Such an extension allows the definitions in the domain model to be inherited.

The *c-reporter*, as defined in document 1.2, represents the specific design framework. It is an abstract component which is described in terms of the behaviors of the attributes: *producer, sorter* and *deliver* and the relationships between them. The relationships between these attributes define the architectural structure of the component. For example, the output of *producer*, *l* and *m* are the input of *sorter* and *deliver* respectively; the output of *sorter,* i.e. *l1*, is the input of *deliver.* Such a structure is also explicitly depicted in terms of a diagram, see also figure 7.4.

The design instance of the C-REPORTER

The design instance of the C-REPORTER is specified in terms of two schemes in the documents 1.3a and 1.3b. The former specifies the subcomponents, the later specifies the composition of them. The specific design instance refers to *nlc-reporter*, the component defined in document 1.3b.

The entities of the design instance of the C-REPORTER are specified in terms of the C-REPORTER-INSTANCE-ATTRIBUTES, a scheme in document 1.3a. The subcomponents *retriever, quick-sorter* and *printer* defined in this scheme, are the instances or values of the attributes *producer, sorter* and *deliver* defined in the scheme C-REPORTER-DOMAIN respectively. The specificity is reflected in the data types which are derived from an instantiation (see object part of the scheme) of the C-REPORTER-FRAME. The actual parameters are prepared by the schemes NUMB, KEY, and LINE at the beginning of document 1.3a. Moreover, the

Document 1.1. The domain model of the C-REPORTER.

```
scheme ELEM = class type Elem end
scheme LIST (E:ELEM) = class type
      List == empty | add (head:E:Elem, tail:List) end
scheme DESC (E:ELEM) = class type
      Desc == empty | add (head:E:Elem, tail:Desc) end
scheme FILE (E:ELEM) = class type
      File == empty | add (head:E:Elem, tail:File) end
scheme C-REPORTER-MODEL(E1: ELEM, E2:ELEM, E3:ELEM )=
class
   object
        D: DESC(E1),
        L: LIST(E2),
        F: FILE(E3)
   type
        Desc = D.Desc,
        List = L.List,
        File = F.File,
        Message
   value
        empty: <>
        producer: Desc ⟶ (List × Message),
        sorter: List ⟶ List,
        deliver: (List × Message) ⟶ File
   axiom forall d: Desc, l,l1: List, f:File, m:Message •
        producer(d) as (l,m)
              pre d ≠ empty
        sorter(l) as l1
              post card elems l = card elems l1
        deliver(l, m) as f
              pre sorter(l)=l
end
```

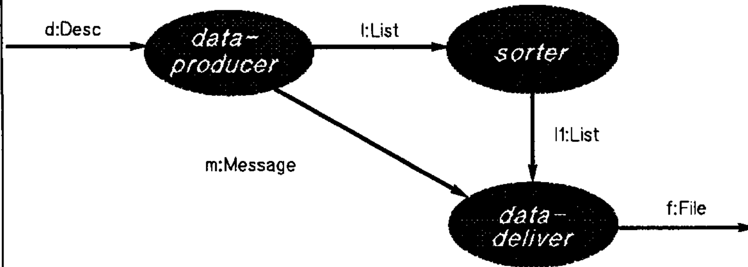
specificity is also reflected in the axiom part in which the properties of *quick-sorter* are precisely provided, although a refinement is still necessary for *retriever* and *printer.*

The structure of the design instance in the C-REPORTER is specified in

Document 1.2. The design framework of the C-REPORTER.

```
scheme C-REPORTER-FRAME(E1: ELEM, E2:ELEM, E3:ELEM)=
extend C-REPORTER-MODEL(E1, E2, E3) with
   value
        c-reporter: Desc ⟶ File,
   axiom forall d: Desc, f:File •
        c-reporter(d) as f
             post ∃ l,l1:List, m:Message•
                  producer(d)=(l,m) ∧
                  sorter(l)=l1 ∧
                  deliver(l1,m)=f
end
```

Figure 7.4. The Design Framework of the C-REPORTER.



document 1.3b in terms of the scheme C-REPORTER-INSTANCE-SUBSYSTEM. Such a scheme is an extension of the entity specification in document 1.3a. As the entities are already specified, the C-REPORTER-INSTANCE-SUBSYSTEM captures only the composition of the entities, resulting in *nlc-reporter*, a

Document 1.3a. The entities of the design instance of the C-REPORTER.

```
scheme NUMB = class type Numb == Real | Int | Nat | – end
scheme KEY = class type Key = Text end
scheme LINE = class type
        eps-Line, pic-Line,
        txt-Line = Text,
        Line == eps-Line | pic-Line | txt-Line
end
scheme
C-REPORTER-INSTANCE-ENTITIES ( K:KEY, N:NUMB, L:LINE )=
class
   object
        DF: C-REPORTER-FRAME(K{Key for Elem},
                             N{Numb for Elem}, L{Line for Elem})
   type
        KeyList = DF.Desc
        NumList = DF.List
        File    = DF.File
        Message = DF.Message
   value
        retriever:   KeyList ⟶ (NumList × Message),
        quick-sorter: NumList ⟶ NumList,
        printer:     (NumList × Message) ⟶ File
   axiom  forall k: KeyList, l,l1:NumList, m: Message
        retriever(k) ≡ DF.producer(k)
        quick-sorter(l) as l1
              ∀i, j : Nat• {i, j} ⊆ inds l1 ∧ i < j ⇒ l1(i) ≤ l1(j)∧
              ∀e : List• card{i | i : Nat • i ∈ inds l ∧ l(i) = e} =
                     card{j | j : Nat • j ∈ inds l1 ∧ l1(j) = e}
        printer(l,m) ≡ DF.deliver(l,m)
end
```

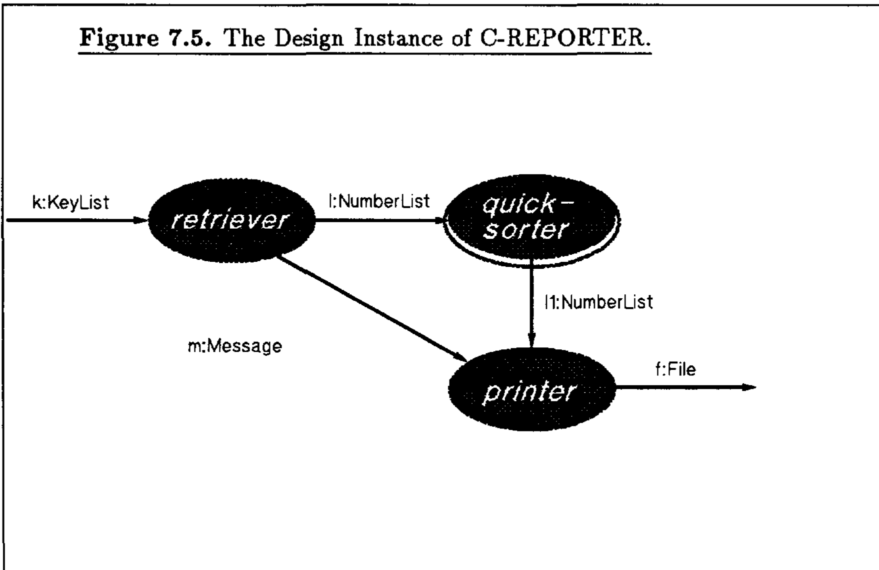
compositive component. The structure of *nlc-reporter* is reflected in the relationships between the entities, as described in the axiom part of the specification. Such a structure is also depicted in terms of a diagram in figure 7.5.

Notice that although the design instance of the C-REPORTER is specific in contrast with the corresponding design framework, the subcomponents

Document 1.3b. The design instance of the C-REPORTER.

```
scheme
C-REPORTER-INSTANCE-SUBSYSTEM ( E:ELEM, N:NUMB, L:LINE)=
extend C-REPORTER-INSTANCE-ENTITIES(E, N, L) with
  value
      nlc-reporter: KeyList ⟶ File,
  axiom forall k: KeyList, f:File •
      nlc-reporter(k) as f
          post ∃ l,l1:NumList, m:Message •
              retriever(k)=(l,m) ∧
              quick-sorter(l)=l1 ∧
                printer(l1,m)=f
end
```

Figure 7.5. The Design Instance of C-REPORTER.



contained in it can be abstract in contrast with their refinements which will be described at lower level. Such an abstraction is reflected not only in the parameter of the scheme, including E:ELEM, N:NUMBER, L:LINE, but also in the partially provided semantics of the subcomponents, such as that of *retriever* and *printer* as shown in document 1.3a.

7.2.2 Specifying the PRINTER (A second level node)

In this subsection, PRINTER, the refinement of the *printer* in the top-node C-PRINTER, is specified in four text files: document 2.1, document 2.2, document 2.3a and document 2.3b. Document 2.1 contains the domain model of the PRINTER, document 2.2 contains the design framework of the PRINTER, and documents 2.3a and 2.3b contain the design instance of the PRINTER, including the subcomponents the design instance consists of and the structure the design instance is composed with.

The domain model of the PRINTER

The domain model of the PRINTER aims at specifying the attributes (abstract subcomponents), which are used to form the design framework of the PRINTER, including *analyze, catalog* and *graph.*

The domain model of the PRINTER is specified in terms of the PRINTER-MODEL, a scheme in document 2.1. The attributes of the domain model are defined in the scheme as function spaces, identified by values with the same names.

The abstraction of the attributes is reflected in two dimensions. First, the PRINTER-MODEL is a parameterized scheme. The parameters as defined in scheme RECEIVED-NUMBER, INTER-RECORD and FILE-LINE determine the abstraction of the attributes being specified. Besides, the axioms of the scheme also make the attributes abstract, since the properties (or constraints) on the subcomponents are only partially described as shown in the axiom part of the scheme. The abstraction is important since only by such an abstraction the design framework can be a framework so as to be reused in constructing various design instances.

Additionally, only for technical reasons, some schemes defined in document 1.1 are reused as type definitions as shown in the *object* part of the PRINTER-MODEL, instead of defining them redundantly.

The design framework of the PRINTER

The design framework of the PRINTER is specified in terms of the PRINTER-FRAME, the scheme in document 2.2. In this scheme, the *printer* is defined as a function space, a composition of the attributes defined in the PRINTER-MODEL. The architectural structure of the composition is also explicitly

Document 2.1. The domain model of the PRINTER.

```
scheme RECEIVED-NUMBER = class ReceivedNumber end
scheme INTER-RECORD = class type InterRecord end
scheme  FILE-LINE = class type FileLine end
scheme PRINTER-MODEL( RN:RECEIVED-NUMBER,
                IR:INTERIM-RECORD, FL: FILE-LINE )=
class
    object
        L:LIST(RN),
        R:LIST(IR),
        F:FILE(FL)
    type
        NumList=L.List,
        File=F:File,
        RecList=R.List,
        Mode
        Message=Text
    value
        analyze: Message × NumList ⟶ Mode × NumList,
        catalog: NumList ⟶ RecList,
        graph:  Mode × RecList ⟶ File
    axiom forall l,l1:NumList, msg: Message, mod:Mode, r:RecList, f:File•
        analyze(msg,l) as (mod,l1),
        catalog(l) as r,
        graph(mod,r) as f
end
```

depicted in figure 7.6.

Additionally, the PRINTER-FRAME is an extension of the PRINTER-MODEL. Such an extension allows the inheritance of all the definitions provided by the PRINTER-MODEL.

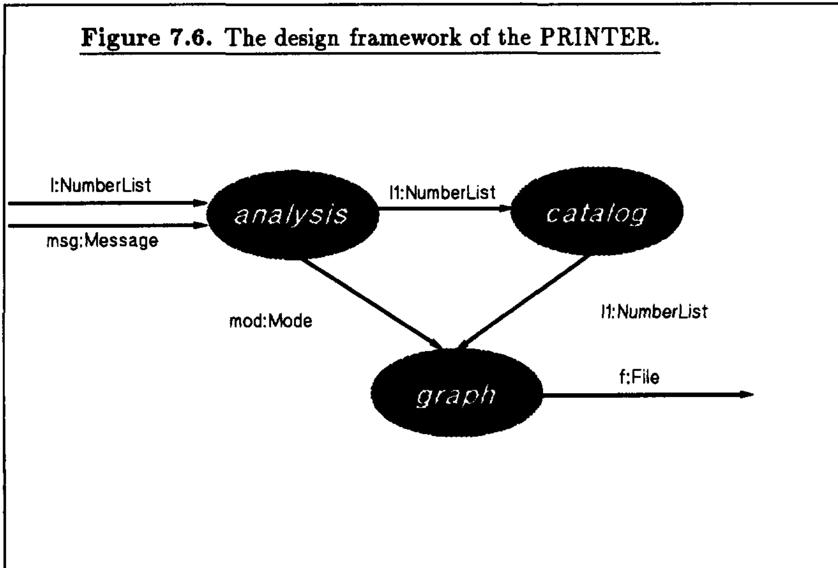
The design instance of the PRINTER

The entities of the design instance of the PRINTER are specified in terms of the PRINTER-INSTANCE-ATTRIBUTES, a scheme in document 2.3a. The subcomponents *verify, grouping* and *chart* defined in this scheme, are the specialization of the attributes *analyze, catalog* and *graph*, as defined in the

Document 2.2. The design framework of the PRINTER.

```
scheme PRINTER-FRAME( RN:RECEIVED-NUMBER,
                      IR:INTERIM-RECORD, FL: FILE-LINE )=
extend PRINTER-MODEL(RN, IR, FL)
    value
        printer: NumList × Message ⟶ File
    axiom forall l:NumList, f:File•
        printer(l) as f
          post ∃ msg:Message, mod:Mode, l1:NumList, r:RecList •
               analyze(msg,l)=(mod,l1) ∧
               catalog(l1)=r ∧
               graph(mod,r)= f
          pre l ≠ empty
end
```

Figure 7.6. The design framework of the PRINTER.



PRINTER-DOMAIN respectively. The specificity is reflected in the data types which are derived from an instantiation (see object part of the scheme) of the PRINTER-FRAME. The actual parameters are prepared by the scheme REAL-NUMBER, GROUP-RECORD AND PIC-LINE at the beginning of document 2.3a. Moreover, the specificity is also reflected in the axiom part,

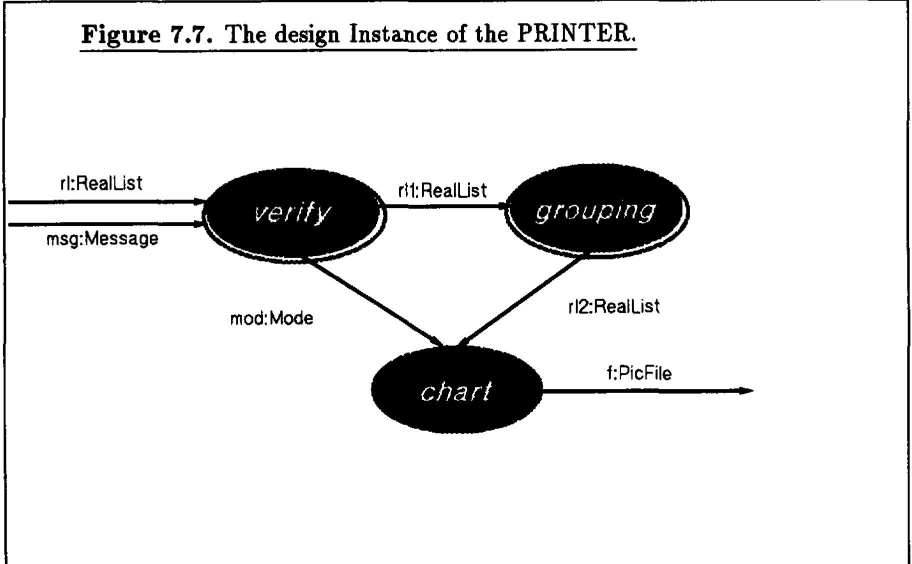
Document 2.3a. The entities of the design instance of the PRINTER.

```
scheme REAL-NUMBER = class type RealNumber=Real end
scheme GROUP-RECORD= class type GroupRecord::
        number: Real, group: Int, descript: Text, color: Text end
scheme PIC-LINE = class type PicLine=Text end
scheme PRINTER-INSTANCE-ENTITIES( RN:REAL-NUMBER,
                GR:GROUP-RECORD, PL: PIC-LINE )=
   object
      PF: PRINTER-FRAME(RN{ RealNumber for Numb},
                GR{ InterRecord for Record}, PL{ PicLine for Line})
   type
      RealList = PF.NumList,
      RecList = PF.RecList,
      PicFile = PF.File,
      Message = PF.Message
      Mode    = Nat
   value
      verify: RealList × Message ⟶ RealList × Mode,
      grouping: RealList ⟶ RecList,
      Chart: RecList × Mode ⟶ PicFile
   axiom forall nl:RealList, msg:Message, mod:Mode,
                rl: RecList, pf:PicFile, x: Int •
      verify(nl,msg) ≡
            if  hd nl > 0
            then case msg of
                  "block-chart" ⟶ (nl, 1),
                  "circular-chart' ⟶ (nl, 2)
            e verify( tl nl, msg) end
      grouping(nl)  as rl,
           post ∀ i  in inds nl •
                  i  in  inds rl ∧
                  rl(i).number = nl(i) ∧
                  rl(i).group =  it nl(i) ∧
                  rl(i).descript = "group" ∧
                  rl(i).color =  if (i/2 * 2 = i)
                                then "GREEN" else "READ"
      chart(rl) as pf
end
```

Document 2.3b. The design instance of the PRINTER.

```
scheme
PRINTER-INSTANCE-SUBSYSTEM( RN:RECEIVED-NUMBER,
                    IR:INTERIM-RECORD, PL: PIC-LINE )=
extend PRINTER-INSTANCE-ENTITIES( RN, IR, PL) with
    value
        PicPrinter: RealList × Message ⟶ PicFile
    axiom forall nl:RealList, msg:Message, pf:PicFile•
        PicPrinter(nl, msg) as pf
            post ∃ mod:Mode, rl:RecList•
                verify(nl, msg) = (nl, mod) ∧
                grouping(nl) = rl ∧
                chart(rl,mod) = pf
            pre  nl ≠ empty
end
```

Figure 7.7. The design Instance of the PRINTER.



in which the properties of the subcomponent *verify* and *grouping* are precisely provided, although a refinement is still necessary for the *chart*.

The design instance of the PRINTER is specified in terms of the PRINTER-

INSTANCE-SUBSYSTEM, a scheme in document 2.3.b. Such a specification is an extension of the PRINTER-INSTANCE-ENTITIES as shown in document 2.3a.

In this scheme PicPrinter stands for the specific design instance, a composition of the entities (subcomponents) defined in the PRINTER-INSTANCE-ENTITIES. The structure of the design instance is described in the axiom part of the scheme, in which the relationships between the subcomponents can be seen. Such a structure is further depicted in figure 7.7., in which the primitive subcomponents such as *verify* and *grouping* can be identified by the circled nodes of the diagram.

7.2.3 Specifying the RETRIEVER (Another second level node)

The RETRIEVER is the refinement of the non-primitive subcomponent, *printer*, which is defined in the design instance of the C-REPORTER (top-node), see also figure 7.5 and document 1.3a.

In this subsection The RETRIEVER is specified in four text files: document 3.1, document 3.2, document 3.3a and document 3.3b. Document 3.1 contains the domain model of the RETRIEVER, document 3.2 contains the design framework of the RETRIEVER, and documents 3.3a and 3.3b contain the design instance of the RETRIEVER, including the subcomponents the design instance consists of and the structure the design instance is composed with.

The domain model of The RETRIEVER

The domain model of the RETRIEVER aims at specifying the attributes (abstract subcomponents) which are used to form the design framework of the RETRIEVER, including *legality, translator* and *search.*

The domain model of the RETRIEVER is specified in terms of the RETRIEVER-MODEL, a scheme in document 3.1. The attributes of the domain model are defined in this scheme as function spaces,thatothat identified by values with the same names.

The abstraction of the attributes is reflected in two dimensions. First, the RETRIEVER-MODEL is a parameterized scheme. The parameters as defined in schemes TERM, CLAUSE and SQUERY determine the abstraction of the attributes being specified. Besides, the axioms of the scheme also make the attributes abstract, since the properties (or constraints) of the subcomponents

Document 3.1. The domain model of the RETRIEVER.

```
scheme TERM = class Term end
scheme CLAUSE = class Clause end
scheme SQUERY (C:CLAUSE) =
class type
        Squery == empty | add(head:C.Clause, tail:Squery)
end
scheme RETRIEVER-MODEL(N:NUMB, T:TERM, C:CLAUSE)=
class
      object
          D: DESC( T {Term for Elem}),
          L: LIST( N{Numb for Elem}),
          S: SQUERY(C)
      type
          KeyList  = D.KeyList,
          NumList  = L.List,
          Squery = S.Squery,
          Message
      value
          legality:  KeyList   ⟶ KeyList,
          translator: KeyList  ⟶ Squery,
          search:   Squery ⟶ (NumList × Message)
      axiom forall k, k1: KeyList, q:Squery, l:NumList•
          legality(k)  as k1
          transform(k) as q
          search(q)   as l
end
```

are only partially described.

Additionally, in order to reduce redundancy, some schemes defined in document 1.1 are reused here as type definitions appearing in the *object* part of the RETRIEVER-MODEL.

The design framework of the RETRIEVER

The design framework of the RETRIEVER is specified in terms of the RETRIEVER-FRAME, the scheme in document 3.2. In this scheme the

Document 3.2. The design framework of the RETRIEVER.

```
scheme RETRIEVER-FRAME(N:NUMB, T:TERM, C:CLAUSE)=
extend RETRIEVER-MODEL(N, T, C) with
    value
       retriever: KeyList ⟶ NumList
    axiom forall k
       retriever(k) ≡ search(translator(legality(k))))
end
```

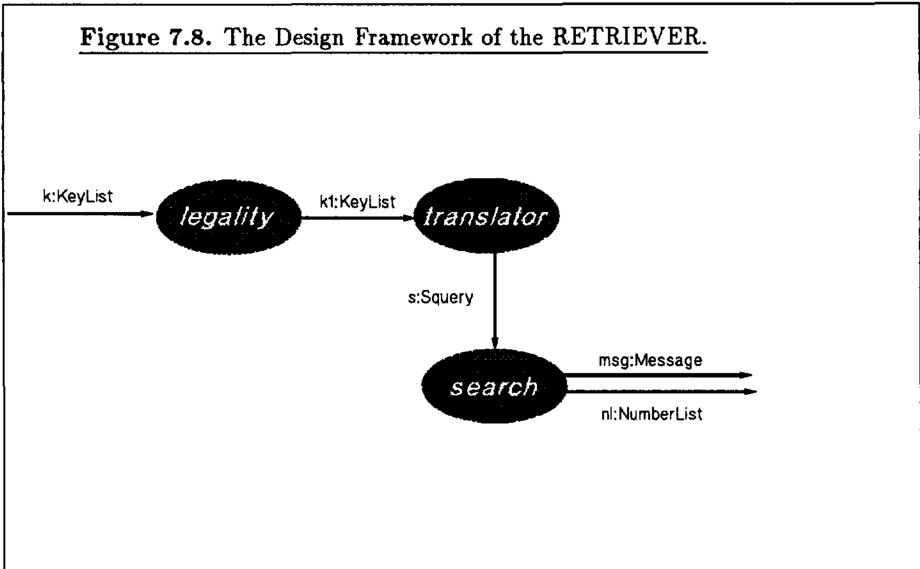


Figure 7.8. The Design Framework of the RETRIEVER.

printer is defined as a function space, a composition of the attributes defined in the RETRIEVER-MODEL. The architectural structure of the composition is also explicitly depicted in figure 7.8.

Additionally, the RETRIEVER-FRAME is an extension of the RETRIEVER-MODEL. Such an extension allows to inherit all the definitions provided by the RETRIEVER-MODEL.

Document 3.3a. The entities of the design instance of the RETRIEVER.

```
scheme INST-NUMB   = class type InstNumb = Real end
scheme INST-TERM   = class type InstTerm = Text end
scheme INST-CLAUSE = class type
                  InstClause ::
                             Label:Nat
                              Expr:Text
                              Div:Char
                 end
scheme
RETRIEVER-INSTANCE-ENTITIES(N:INST-NUMB,
                         T:INST-TERM, C:INST-CLAUSE)=
      object
          R: RETRIEVER-FRAME( N{InstNumb for Numb},
                  T{InstTerm for Term}, C{InstClause for Clause} )
      type
          Spec     = R.KeyList,
          Squery   = R.Squery,
          RealList = R.NumList,
          Message  = Text
      value
          checking: Spec ⟶ Spec,
          remodel:  Spec ⟶ Squery,
          acquire:  Squery ⟶ NumList × Message,
      axiom forall sp, sp1: Spec, sq:Squery, l:NumList, msg:Message •
          checking(sp) as sp1
          remodel(sp) as sq
          acquire(sq) as (l,msg)
end
```

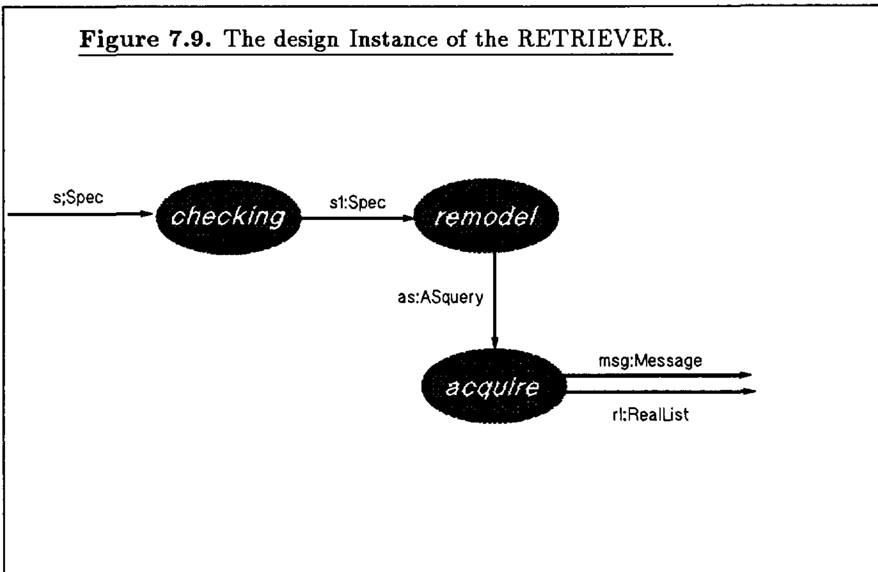
The design instance of the RETRIEVER

The entities of the design instance of the RETRIEVER are specified in terms of the RETRIEVER-INSTANCE-ENTITIES, a scheme in document 3.3a. The subcomponents, *checking, remodel* and *acquire* in this scheme, are the specialization of the attributes *legality, translator* and *search* in the RETRIEVER-DOMAIN respectively. The specificity is reflected in the data types which are derived from an instantiation (see object part of the scheme) of RETRIEVER-

Document 3.3b. The design instance of the RETRIEVER.

```
scheme
RETRIEVER-INSTANCE-SUBSYSTEM(
                    N:INST-NUMB, T:INST-TERM, C:INST-CLAUSE)=
extend RETRIEVER-INSTANCE-ENTITIES (N, T, C) with
      value
          fetcher: Spec ⟶ NumList × Message
      axiom forall s:Spec,
          fetcher(s) ≡ acquire(remodel(checking(Spec)))
end
```

Figure 7.9. The design Instance of the RETRIEVER.



FRAME. The actual parameters are prepared by the scheme INST-NUMB, INST-TERM and INST-CLAUSE at the beginning of document 3.3a. However, the entities of the design instance are not primitive subcomponents. The constraints (or properties) of these subcomponents are slightly described, which will be supplemented in their refinements.

The design instance of the RETRIEVER is specified in terms of the RETRIEVER-INSTANCE-SUBSYSTEM, a scheme in document 3.3b. Such a specification is an extension of the RETRIEVER-INSTANCE-ENTITIES in

document 3.3a.

In this scheme, the *fetcher* stands for the specific design instance, a composition of the entities (subcomponents) which are defined in the RETRIEVER-INSTANCE-ENTITIES. The structure of the design instance is described in the axiom part of the scheme, in which the relationships between the subcomponents can be seen. Such a structure is further depicted in figure 7.9 explicitly.

7.3 Browsing and manipulation

A large-scale component is not only a piece of static data, but a dynamically changing infrastructure. Both the structure and the information of a large-scale component can be browsed and manipulated (modofication and reconstruction). In this section, we discuss how to browse a large-scale component and how to manipulate the component in terms of some tools.

7.3.1 Browsing a large-scale component

Browsing a large-scale component aims at looking through the information contained in the large-scale component, particularly, at looking through the nodes of the H-structure of the large-scale component. The browsing is started from viewing the window at a terminal which displays the information contained in the top-node of the H-structure. The browsing is continued by viewing the window which displays a lower-level node of the large scale component. Alternatively, a parent node can always be traced back if the current node is not top-node.

Browsing a large-scale component from one node to another can be realized by clicking some keyword of the current node. In order to view the refinement of a non-primitive subcomponent of the current design instance, one needs to click the name of the non-primitive subcomponent which appears in the diagram of the design instance. In order to trace back to the parent node, one needs to click the keyword *parent* which appears in the window of the current node.

In order to illustrate how a large-scale component can be browsed practically and what information can be viewed during the browsing, there are three pictures provided, see also figure I1, I2 and I3. Each picture shows an envisaged interface between a node of an H-structure and users. Each

picture consists of four parts, namely, SPEC (top-left), FRAME (top-right), INSTANCE (bottom-left) and DOMAIN (bottom-right). The SPEC part contains documents of specification and source code regarding the design instance, the design framework, and the domain model of the node. The FRAME part contains a diagram, describing the structure of the design framework. The INSTANCE part contains another diagram, describing the structure of the design instance. The DOMAIN part contains domain entities which can be used for the instantiation of the design framework.

The first picture, see also figure I1, displays the C-REPORTER, the top-node of a large-scale component.

In the SPEC part of the C-REPORTER, there are four documents: C-REPORTER-INSTANCE, C-REPORTER-FRAME, C-REPORTER-MODEL and SOURCE-CODE. The document C-REPORTER-INSTANCE contains the specification of the design instance in RSL. The document C-REPORTER-FRAME contains the specification of the design framework in RSL. The document C-REPORTER-MODEL contains the specification of the domain model

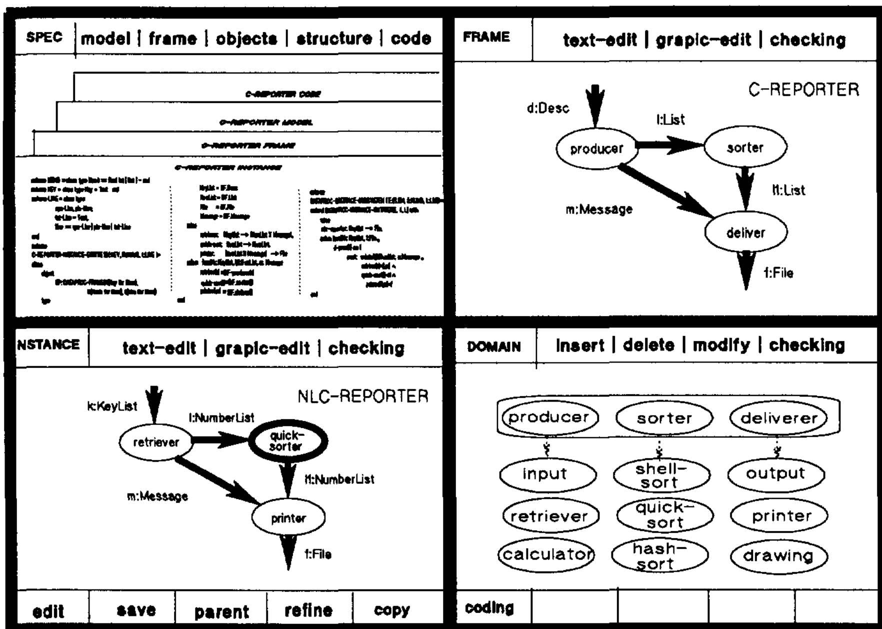


Figure I1. The user interface of the C-REPORTER.

which specifies the abstract subcomponents appearing in *c-reporter* in RSL.

The document SOURCE-CODE contains an implementation of the design instance in the C programming language.

In the FRAME part of the C-REPORTER there is a diagram representing the structure of the design framework, the C-REPORTER, which consists of three abstract subcomponents and the relationships between them. The subcomponents are *producer, sorter* and *deliverer.* The *producer* receives data *d:Desc* from some external component and sends data *l:List* and *m:Message* to the subcomponents *sorter* and *deliverer* respectively. The *sorter* receives data *l:List* and sends data *l1:List* to deliverer. The *deliverer* receives data *m:Message* and *l1:List* and sends data *f:File* to some other external component. The diagram corresponds to the semantic specification contained in the document C-REPORTER-FRAME.

In the INSTANCE part of the C-REPORTER there is a diagram, namely NCL-REPORTER, which depicts the structure of the design instance. NCL-REPORTER consists of three subcomponents and the relationships between them. The subcomponents are *retriever, quick-sorter* and *printer.* The *retriever* receives data *k:KeyList* from some external component and sends data *l:NumbList* and *m:Message* to the subcomponents *quick-sorter* and *printer* respectively. The *sorter* receives data *l:NumList* and sends data *l1:NumList* to *printer.* The *printer* receives data *m:Message* and *l1:NumList* and sends data *f:File* to some other external component. The diagram is corresponding to the semantic specification contained in the document C-REPORTER-INSTANCE. From the diagram of the design instance we can also tell the difference between primitive subcomponents and non-primitive ones. The former is characterized by circles with a thick line, the latter, by a circle with a thin line.

In the DOMAIN part of the C-REPORTER, there is a table. The title of the table consists of three attributes: *producer, sorter* and *deliverer,* corresponding to the abstract subcomponents in the diagram of the FRAME part. Under each attribute, there are domain entities (specific subcomponents) which represent the values of the attribute. For example, *shell-sorter, quick-sorter* and *hash-sorter* are the values of the attribute *sorter.* Moreover, in each row of the table a domain instance is provided. A domain instance consists of a set of domain entities which can be used for the instantiation of the design framework consistently. For example, *retriever, quick-sorter* and *printer* form a domain instance which can be used for the instantiation of the design framework *C-REPORTER.* Notice that the instantiation of a design framework with a domain instance is always consistent, while there is no guarantee

for consistency if arbitrary domain entities are selected for the instantiation, although it is often possible.

From the specification of the design instance NCL-REPORTER, we see that the semantics of the subcomponents *retriever* and *printer* are not provided sufficiently. This is because these subcomponents are non-primitive subcomponents and their semantics will be further supplemented in their refinement. As discussed before, their refinements can be displayed by clicking their names in the diagram of the design instance.

Suppose that we have clicked the name *printer*, a lower-level large-scale component PRINTER is displayed, see also figure I2. As the PRINTER is also a large-scale component, the organizational structure of figure I2 is similar to that of figure I1.

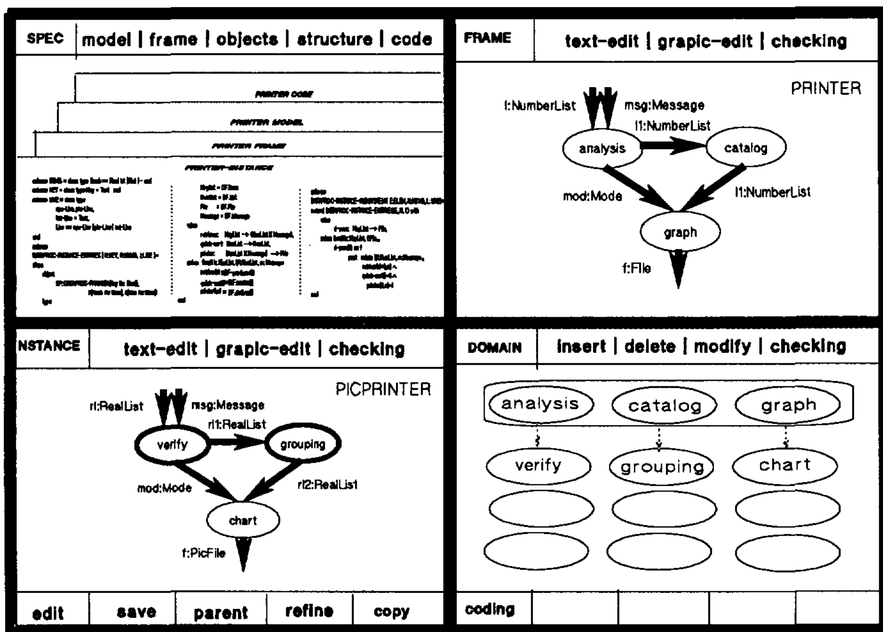


Figure I2. The user interface of the PRINTER.

In the SPEC part of the PRINTER there are several documents in which the schemes PRINTER-INSTANCE, PRINTER-FRAME and PRINTER-MODEL are specified in terms of RSL . Additionally, the document SOURCE-CODE is provided in terms of the C programming language.

In the FRAME part of the PRINTER a diagram is displayed, which con-

sists of three subcomponents and the relationships between them. The *analysis* receives data *l:NumList* and *msg:Message* from external components and sends data *l1:List* and *mod:Mode* to subcomponents *catalog* and *graph* respectively. The *graph* receives data *mod:Mode* and *l1:NumList* and sends data *f:File* to an external component.

Since the PRINTER is the refinement of the *printer* in the higher-level design instance, the design framework of the PRINTER must be consistent with that of the *printer*. From the diagram of the design framework we see that the external input (the data received by *analysis*) and the external output (the data sent to other components by *graph*) are consistent with those of the *printer* in the higher-level design instance. Moreover, the semantic specification of the PRINTER must inherit the related specification from the higher-level design instance. For example, the data *l:NumList* received by *analysis* is not allowed to be empty and must have been sorted, i.e. *pre* $l \neq$ *empty* $\wedge$ *sorter(l)* $= l$, see also document 1.3a and document 1.1.

In the INSTANCE part of the PRINTER, the PICPRINTER, the structure of the design instance is depicted, which is an instance of the design framework. The PICPRINTER consists of the subcomponents: *verify, grouping* and *chart*. The *verify* receives data *rl:RealList* and *msg:Message* and sends *rl:List* and *mod:Mode* to subcomponents *grouping* and *chart*. The subcomponent *grouping* receives data *rl1:RealList* and sends data *rl2:RealList* to *chart*. The subcomponent *chart* receives data *mod:Mode* and *rl2:RealList* from *verify* and *grouping* respectively, and sends data *f:PicFile* to an external component.

Comparing the diagram and the formal specification of the design instance of this node with that of the parent node, we see that the semantics of the PICPRINTER (the current design instance) is consistent with that of *printer*, the non-primitive subcomponent which is refined by the PRINTER, to current node.

In the DOMAIN part of the node PRINTER, there are three domain instances. One of them has been used to construct the current design instance.

It is similar to the way of viewing the node PRINTER, the node RETRIEVER, a refinement of the *retriever* (the other non-primitive subcomponent of the top-level design instance) can be displayed by clicking the name *retriever* in the top-node. The information contained in the RETRIEVER is shown in figure I3.

In the SPEC part of the RETRIEVER, there are several documents in which the schemes RETRIEVER-INSTANCE, RETRIEVER-FRAME and

RETRIEVER-MODEL are specified in RSL . Additionally, the document SOURCE-CODE is provided in terms of C programming language.

In the FRAME part of the RETRIEVER a diagram is displayed, which consists of three subcomponents, *legality, translator* and *search*, and relationships between them. The subcomponent *legality* receives data *k:KeyList*

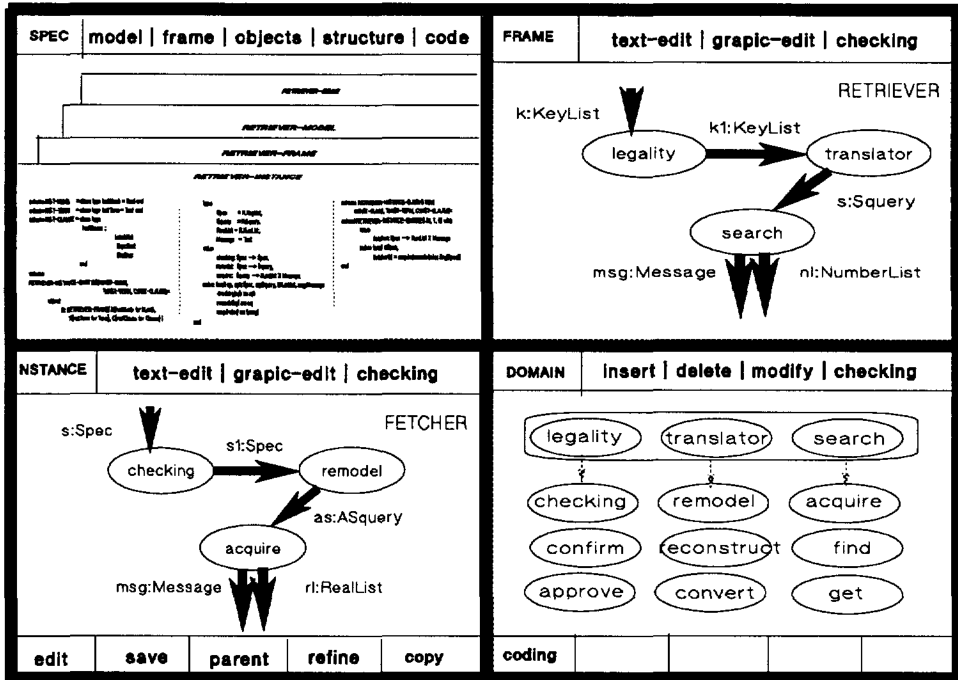


Figure I3. The user interface of the RETRIEVER.

from an external component and sends data *k1:KeyList* to the subcomponents *translator*. The subcomponent *translator* receives data *KeyList* from *legality* and send data *s:Squery* to *search*. The subcomponent *search* retrieves data *s:Squery* from *translator* and sends data *msg:Message* and *nl:NumberList* to external component(s).

It is similar to the PRINTER, the design framework of the RETRIEVER must be consistent with that of the *retriever* in the higher-level design instance. From the diagram of the design framework, we see that the external input (the data received by *legality*) and the external output (the data sent to other components by *search*) are consistent with that of the *printer* in the higher-level design instance.

In the INSTANCE part of the RETRIEVER, FETCHER, the structure of the design instance is depicted, which is an instance of the design framework. The FETCHER consists of subcomponents: *checking, remodel* and *acquire.* The *checking* receives data *s:Spec* and sends *s1:Spec* to *remodel.* The subcomponent *remodel* receives data *s1:Spec* and sends *as:ASquery* to *acquire.* The subcomponent *acquire* receives data *as:ASquire* and sends *msg:Message* and *rl:RealList* to external components.

Comparing the diagram and the formal specification of the design instance of this node and that of the parent node, we see that the semantics of the *fetcher*, the current design instance, is consistent with that of *retriever*, the non-primitive subcomponent which is refined by the RETRIEVER, the current node.

In the DOMAIN part of the node RETRIEVER, there are domain instances. One of them has been used to construct the current design instance.

7.3.2 Manipulating a large-scale component

Manipulating a large-scale component implies creating, deleting or modifying the structure of the component or the information contained in the structure, which can be realized by using the user interface of the component. Since the user interface of a large-scale component is a window which consists of four parts: SPEC, FRAME, INSTANCE and DOMAIN, our discussion will be focused on how each part can be manipulated.

The SPEC part. The SPEC part consists of a set of documents. Each document is a text file. The basic requirements for manipulating these documents are no more than the file operations an operating system has, such as file-creating, file-editing, file-deleting and so on. The activities concerning the SPEC part include

- Programming in a certain programming language, according to the specification and diagram of current design instance; compiling, linking, testing and executing the implementation of current design instance.

- Making a decision about the current version of implementation and specification, if there are several.

- Specifying the domain model, design framework and design instances.

- Verifying the consistency between the different specifications inside the node and between the current specifications and that of parent nodes.

The FRAME part. The FRAME part contains a diagram, representing the structure of the design framework. The basic requirement for manipulating the diagram is a graphic editor or a graphic generation. A graphic editor provides the means to create, deposit or modify the diagram directly. A graphic generator allows to create, deposit and modify the diagrams in terms of textual specification. A graphic editor is more intuitive and flexible than a graphic generator, whereas a graphic generator is economic in repository and automatic in drawing the diagram. The activities concerning the FRAME part include

- Creating a new design framework by the decomposition of a non-primitive subcomponent of the parent node, resulting in a diagram which will be a guideline to define domain resources and a guideline for formal specification.
- Generating a design framework (diagram and specification) by the parameterization of a new design instance.
- Modifying the design framework, concerning the repository of the old node to its parent domain resources optionally.
- Verifying the consistency between the diagram and its semantic specification (in SPEC part). For example, each node in the diagram of the design framework must be corresponding to a function space which is defined in the document for specifying the domain model. Furthermore, if two nodes of the diagram are connected together, such a connection can be identified from the specification of the design framework.
- Carrying out the instantiation of the design framework with domain resources, resulting in a candidate for a design instance (diagram and specification).

The INSTANCE part. The INSTANCE part contains a diagram. The basic requirement of this part is similar to that of the FRAME part.

The INSTANCE part is the most active part in a node of a large-scale component. It is an abstraction of a real design that a system developer needs,

and a guideline to understand the specification (in the document part) of the design, and a guideline for the reconstruction of the design. The activities concerning the INSTANCE part include

- Switching the control to the refinement of a non-primitive subcomponent of the design instance so as to create, review or reconstruct the refinement, or tracing back to the parent node so as to deal with other branches of the parent node.
- Making decisions about the modification of current design instance and replacing one or more subcomponents in the current design instance with domain entities or with newly created ones.
- Making decisions about the instantiation of the design framework with a domain instance from the domain resources, in order to generate an instance of the design framework;
- Making decisions about the specialization of the interim instance, concerning some special requirements of a design instance.

The DOMAIN part. The DOMAIN part displays only the components (large-scale components or code components) which can be used to instantiate the current design framework or to modify the current design instance.

The basic requirements of the DOMAIN part include the means to insert, delete and modify each domain instance or each domain entity of the domain resources. The particular activities to manipulate the domain resources include

- Switching the control to a copy of a particular design entity, to review or reconstruct the design entity into a new one, in order to construct a new design instance so as to instantiate the design framework or modify current design instance.
- Making decisions on the acceptance of design instances by verifying the consistency of this instance with the domain model as specified in the SPEC part, or making decisions on the deletion of a current domain instance.
- Switching the control to existing domain entities so as to view the details of the entity precisely, especially, when a domain entity is a large-scale component.

7.4 Generating alternative designs

The purpose of having large-scale components is to provide an infrastructure so that domain resources can be accumulated, and alternative designs and implementations can be generated with respect to the large-scale component with reuse-in-the-large.

What is an alternative design? As discussed before, a large-scale component has its design instance at each node of its H-structure. The instances at different levels of the H-structure, as a whole from top to primitive nodes, form the design of a large-grain component (subsystem or system) at different levels of abstractions. Therefore, generating an alternative design of a large-grain component from a large-scale component implies the generation of a design which is different from the current design instances. Such an alternative design can be generated by replacing or modifying one or more design instances of the current design. The design instances to be replaced or modified can be either at high-level abstraction or at low-level in the H-structure.

The application domain of a large-scale component A large-scale component aims at developing a set of alternative large-grain components such as subsystems or even complete systems in an application domain. The boundary of the application domain is defined by the top-level design framework of a large-scale component, concerning both the degree of abstraction given by the domain model and the way of their composition as described by the design framework. For example, the designs of all c-reporters which can be developed with the reuse of C-REPORTER must be the instances (objects) of the design framework which is precisely defined by the scheme C-REPORTER-FRAME, see also document 1.2.

Although such a definition is normally broad enough for an institute that needs to develop a large-number of large-grain components within the application domain, the domain can be extended by building higher-level abstractions. For example, we may extend the intention of the *c-reporters* by addressing a more general concept, saying general-c-reporters. This can be done by reconstructing the top-node of the large-scale component. For example we may reconstruct the top-node C-REPORTER into the nodes as shown in figure I4 and I5, leaving all other nodes unmodified.

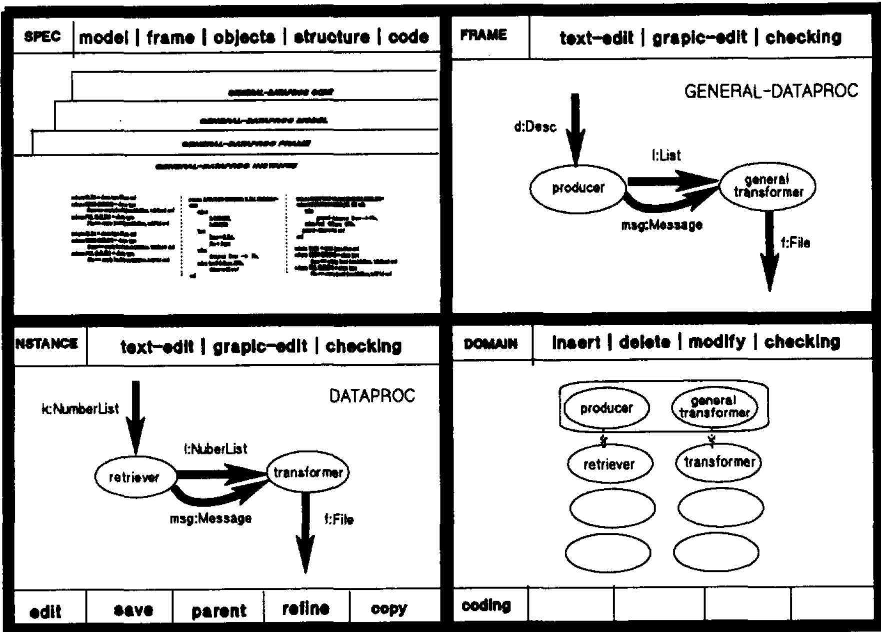


Figure I4. The user interface of GENERAL-C-REPORTER.

Domain analysis as learning. Domain analysis as learning was addressed by Arango[3]. "Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same tasks or tasks drawn from the same population more efficiently and more effectively next time"[110]. The process of reusing large-scale component is a learning process, which allows the H-structure and the information contained in the structure to be changed dynamically, leading to more effective reapplication of the component later on. For example, if there is only one domain instance in the C-REPORTER, we have one choice to construct a c-reporter at top-level with reuse. However, if we reuse only *input* and *shell-sorter*, the first two entities of the domain instance, and create *printer*, the third entity, then a new domain instance is formed. By putting the new domain instance into domain resources, the next reuser will have two choices to construct a c-reporter at the top-level.

There is a great difference between traditional library components and large-scale components. The former are created and managed by different people concerning universal application; the latter, by system developers themselves, concerning an application domain. More importantly, the content of the

traditional library components are normally unchanged after the reuse of the component; while the content of large-scale components can be modified and reconstructed by system developer (reusers). There are few operations which are available to manipulate traditional library components, while a large-scale component and its support environment form an infrastructure.

In fact, a large-scale component is an open component to system developers. It is created by system developers, managed by system developers, manipulated by system developers and reused by system

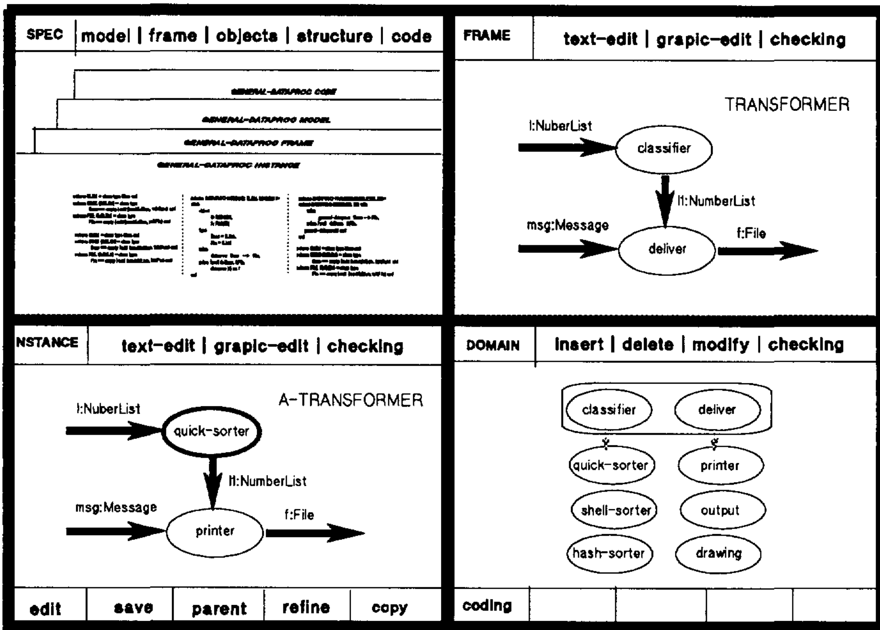


Figure I5. The user interface of TRANSFORMER.

developers. Although the risk of an open component seems the possibility to make it worse after some modification, we think such a risk is practically small for reusing a large-scale component. The wish to have open system components originats from domain analysis. The quality of the domain resources is evaluated by domain analyzers, while the reusers of a large-scale component, who modify the component, are domain analyzer themselves. Moreover, the possibility of semantic verification, executable implementation and the transparent organizational structure of a large-scale component are also useful to guarantee the quality of the domain resources.

7.5 Generating the implementation

Implementation. In a large-scale component, there is always a *current design* which consists of the design instances at different levels of abstraction. The *implementation* of the current design is a large-grain component which is composed from the primitive subcomponents appearing in the design instances at different levels. Such a composition is supported by a large-scale component by the primitive components and the specification of the design instances at different levels. The primitive components provide the source code components which the large-grain component is composed from, while the specification tells us what a large-grain component needs and how the large-grain component can be composed from the primitive components.

As a restriction, the current design must be implemented in the same programming languages or the languages that can be interfaced to each other.

Version control. Although the implementation of the current design must be implemented in terms of the same programming language, different current design instances (at different time) can be implemented in terms of different programming languages. This implies that the large-scale component should support *version control* for its implementation so that the reusability of a single large-scale component is independent of programming languages.

As discussed before, the version control in a large-scale component can be realized by allowing different versions of the implementation to be contained in the SPEC part (for each node of a H-structure) as shown in the window. The different versions might be either the implementation in different languages or the implementation in different manner in the same programming language.

In order to realize the version control, we need to mark one of the implementations as the *active* or *current* version which presents the current implementation of a design. Consequently, during the process of component composition, we must check not only whether a non-primitive subcomponent is implemented but also whether it is implemented in a programming language and whether the proper implementation is marked as the active version.

The process of the composition. The process of composing the implementation for the design instance (starting from top-level) is that if it contains

non-primitive subcomponents which have not been implemented, go to compose the non-primitive subcomponents first; then compose the implementation of the current design instance from the primitive components and the implemented non-primitive subcomponents. The composition will be guided by the specification of the current design instance. For the non-primitive subcomponents, it is needed to switch the control (interface of a large-scale component) from current node to its refinement. The rest activity for the composition is similar to that of a top-level design instance, since a large-scale component is defined recursively.

In order to provide an intuitive impression on the composition process, we describe how an implementation of C-REPORTER can be composed according to the specification discussed before. For such a composition we assume that each primitive subcomponent of the large-scale component is written in C and can be accessed. To simplify the process of such a composition, we also assume that the subcomponents appearing in the *printer* and *retriever* are primitive subcomponents or implemented non-primitive subcomponents.

The example given in this subsection consists of a set of files. The C-code which this file contains is derived from the large-scale component C-REPORTER. We assume that all primitive subcomponents of the large-scale component are provided in terms of C-code.

The *type.h*, a file, contains a group of type definitions which correspond to several type trees that can be shared by all other files.

File *printer.h* contains the domain entities of the design instance PRINTER. File *printer.c* is a composition of the entities in file *printer.h*, which can be done according to the specification of the design instance of the PRINTER.

Similarly, file *retriever.h* contains the domain entities of the design instance of the PRINTER. File *retriever.c* is the composition of the entities which are contained in file *retriever.h.* The composition can be done according to the specification of the design instance in the RETRIEVER. From the *printer.c* and *retriever.c*, *c-reporter.h* has been made, corresponding to the domain entities appeared in the design instance of C-REPORTER. From the contents of the *c-reporter* the large-grain component *c-reporter.c* has been composed, according to the C-REPORTER-INSTANCE-SUBSYSTEM, the specification of the design instance of the C-REPORTER.

TYPE-COLLECTION (type.h)

```
#define make-type(type, list) struct type { list };
      typedef struct type *type
make-type (NumMod, RealList  nl; Mode  mod)
make-type (NumMsg, RealList  nl; Message  msg)
make-type (Rec, Real  numb; int  group; Text
                  descript; Text  Color)
typedef char *Text;
typedef float Real;
typedef int   Mode;
typedef Text  Message;
typedef Text  InstTerm;
typedef Text  PicLine;
typedef Rec     *RecList;
typedef Number  *RealList;
typedef Text    *Spec;
typedef Text    *Squery;
typedef PicLine *PicFile;
```

PRINTER-INSTANCE-ENTITIES (printer.h)

```
NumMod      verify (RealList; Message);
RecList     grouping (RealList);
PicFile     chart (Mode; RecList);
```

PRINTER-INSTANCE-SUBSYSTEM (printer.c)

```
#include "type.h"
#include "printer.h"
PicFile printer (RealList  nl; Message msg)
{
      NumMod  nm;
      RecList  rl;
      nm = verify (nl, msg);
      rl = grouping (nm→nl);
      return ( chart ( rl, nm→mod ) );
}
```

RETRIEVER-INSTANCE-ENTITIES (retriever.h)

```
Spec        checking (Spec);
Squery      remodel (Spec);
NumMsg      acquire  (Squery);
```

RETRIEVER-INSTANCE-SUBSYSTEM (retriever.c)

```
#include "type.h"
#include "retriever.h"
NumMsg fetcher (Spec s)
{
    return ( acquire ( remodel ( checking (s) ) ) );
}
```

C-REPORTER-INSTANCE-ENTITIES (c-reporter.h)

```
    NumMsg  fetcher (Spec);
    NumList quick-sorter (NumList);
```

7.6 Summary

In this chapter we discussed how reuse-in-the-large happens in a support environment, regarding a *Student Management System* (SMS), an application domain of reuse-in-the-large.

In section 1 we briefly described SMS, the Student Management System. In section 2 our discussion was focused on the C-REPORTER, a subsystem of the SMS. Particularly, we discussed the representation of the C-REPORTER, including both textual and diagrammatic specification of the component. In section 3 we described the interface between a system developer and a large-scale component, i.e. the C-REPORTER, with an assumption that the C-REPORTER has been reused and a set of domain resources exist. The example

in this section shows us that a large-scale component is an infrastructure for the development of a set of similar components in an application domain. The constituent parts of a large-scale component are not only transparent but also manipulatable to system developers.

C-REPORTER-INSTANCE-SUBSYSTEM (c-reporter.c)

```
#include "type.h"
#include "retriever.h"
#include "printer.h"
PicFile nlc-reporter ( Spec s )
{
        NumMsg nm;
        NumMod mod;
        nm = fetcher (s);
        nl = quick-sorter (nm→nl);
        return ( printer ( nl, nm→msg));
}
```

In section 4 using the C-REPORTER, we illustrated how to generate an alternative design (instance) from a large-scale component and how to collect the resources of a large-scale component. An alternative design is the design of a particular (large-grain) component to be made by the system developer. In section 5 we demonstrated how a particular (large-grain) code component can be generated when the design of the component is complete as illustrated in section 4.

From the prototyped example and the discussion concerning the example, we conclude that, in terms of reuse-in-the-large, (1) an application developer is capable of reusing not only the source code but also the design information of large-grain components; (2) an application maintainer is capable of maintaining a large-grain component with the manipulation of a large-scale component, especially with the modification of the design instance. In this case the maintenance of an application is equivalent to the development of the application with reuse-in-the-large, and vice versa.

Chapter 8

Epilogue

We now have the second software crisis: maintenance.
- G. Cruman, 1987.

This dissertation has been carried out in the context of two other research topics: *the management of information systems*[82] and *dynamic modelling of information systems*[113]. On the one hand, this approach was embedded in the research on the management of information systems, aiming at supporting application maintenance; on the other hand, such an approach is a means for dynamic modelling of applications, aiming at meeting the novel requirements for system development. In order to reach both aims, the technology for application reuse plays a leading role, resulting in an environment to support application development and maintenance, i.e. an information system for application reuse-in-the-large.

8.1 Reuse-in-the-large

From a reuse point of view, this dissertation has provided a solution for application reuse-in-the-large. According to this work, reuse-in-the-large is not only an attractive issue but also an applicable means to automate the process of application development and maintenance on a large-scale. For the realization of reuse-in-the-large, the concept of large-scale components was addressed, concerning an integrated representation of design frameworks, design instances and domain resources for a set of applications in an application

domain. For such a representation a pragmatic model was proposed, which allows the information of a large-scale component to be represented at different levels of abstraction. For the manipulation of large-scale components an operating model was described, covering the activities of application analysis, design, implementation and maintenance. For the management of large-scale components, a semantic data base management system was investigated, concerning the repository and retrieval of large-scale components. For the transparency of large-scale components to users and for supporting application development and maintenance with reuse-in-the-large, a user interface was prototyped, concerning creation, browsing, reconstruction and reuse of large-scale components.

8.2 Supporting management of information systems

The research on management of information system, as described in [82], provides a management framework for the support and maintenance of operational *information systems.* An operational information system was defined as a system for collecting, storing and processing data.

The research on management of information systems can be divided into three sub-fields: *functional management, application management,* and *technical management.* Functional management is concerned with functional maintenance and user support and is related to the use and functionality of hardware, software and data bases. Application management is concerned with the tasks of application software maintenance and the technical support for such tasks, including the facilities for the automation of application maintenance. Technical management is concerned with the tasks of operational control and the support for such tasks, including technical support for hardware, system software, communication facilities, data base management systems, data banks and application software which are operational and in use, including also the technical services for daily use and for planned use.

Obviously, management of information systems is a broad research field, involving almost all techniques and facilities applied in information systems. For instance, the *state model*[82], a management framework of information systems as in figure 8.1, involves the whole life-cycle of information systems, which is expected to define, analyze, evaluate and control management or-

ganization in real world situation. According to this model, the life-cycle of an information system consists of several states: *development*, regarding the analysis, design and implementation of an information system, *acceptance*, regarding the test on performance and functionality of an information system, *exploitation*, regarding the installation and execution of an information system, *use*, regarding the activities to solve users' problems, and *maintenance*, regarding the modification of information systems after delivery.

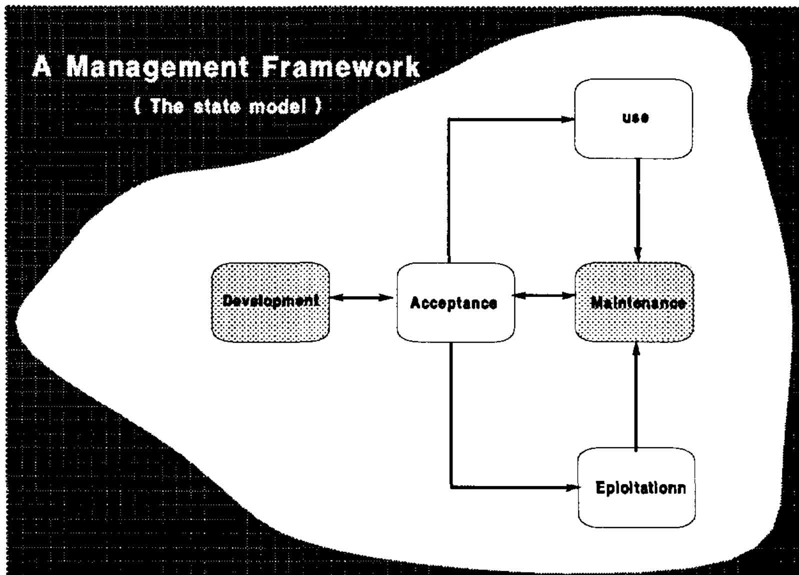


Figure 8.1. The state model of management information systems [1].

From the management of information systems point of view, the information system for application reuse-in-the-large, as described in this dissertation, is an information system which provides an integrated support for application development and maintenance, regarding two states of the state model in figure 8.1, i.e. *development* and *maintenance*.

For application development, a large-scale component plays the role of domain analysis for application construction. An application developer may design a new application by reusing both the process of design and the artifacts which represent the design, analysis and implementation of existing applications.

[1]Figure 8.1 is derived from a similar figure appearing in [81].

For application maintenance, a large-scale component provides an infrastructure for the understanding and reconstruction of existing applications. The maintenance can be corrective maintenance, adaptive maintenance, additive maintenance and so on, aiming at generating alternative applications in an application domain.

Maintenance, according to our approach, implies not only the modification and reconstruction of existing applications, but also of existing designs. The advantage of dealing with existing designs is the possibility of allowing maintenance to be done at abstract (design rather than code) levels. This is especially important for the maintenance of large and complex applications.

According to our approach, application maintenance can be done at different levels of designs. The artifacts to be reused for maintenance at each level include design instances, design frameworks and domain resources. Moreover, the process of creating these artifacts, i.e. the process of application development, can also be reused. The process of application development can be recovered from the H-structure of a large-scale component. To the whole H-structure an application maintainer can trace the process of design decomposition top-down and the process of composition bottom-up from the structure. To each node of the H-structure an application maintainer may find the information about how the design is constructed, how to build alternative designs and alternative implementations, and what resources are available to construct the alternatives.

8.3 Supporting dynamic modelling of applications

As discussed at the beginning of this dissertation, the novel requirements for system development in dynamic modelling, have been the basis for the research setting of the dissertation work. According to the philosophy of dynamic modelling[113], we need an environment which supports the problem solving process of application development.

In this dissertation work the support environment corresponds to an information system for application reuse-in-the-large. The problem solving process is what Wegner[125] called a uniformed problem solving process, i.e. the process of analysis, design, implementation and maintenance. The novel requirements are what Sol[113] described as the need for a *common frame* of reference, concerning a coherent set of descriptive building blocks, the need for an *extendible* system description and analysis context, the need for an *it-*

erative process of analysis and synthesis, and the need for the generation of *alternative* possibilities.

In order to meet the novel requirements, design frameworks have been defined as common frames of reference for the composition of alternative designs in an application domain. Such a common frame is concerned with a coherent set of domain resources. The domain resources are lower-level entities which can be used for the composition of higher-level designs.

In order to support an extendible application description and analysis context, the H-structure was used for the representation of large-scale components. An H-structure is a dynamic structure which can be extended from representing a single application to representing a domain analysis incrementally. A domain analysis is a basis to compose applications with reuse-in-the-large. Investigating the domain analysis, an application developer may learn application design from experience. Browsing the H-structure, an application designer may understand the analysis context for a particular system component.

In order to support the iterative process of analysis and synthesis, large-scale components have been designed by an integration of the design framework with design instances. Such an integration allows application developers to experimentally complete their analysis and synthesis, which was typically described in the process of application prototyping. The process of analysis refers to the process of top-down decomposition, and the process of synthesis refers to the process of reusing lower-level entities for the construction of the higher-level designs and implementations.

In order to support the generation of alternative possibilities, flexibility is provided for modifying design instances, reconstructing design frameworks and updating domain resources. An alternative design can be generated once a design instance of the H-structure is changed, resulting in varieties of applications with alternative functions and performance.

The future research on reuse-in-the-large may be concerned with how to extend the possibility of reuse-in-the-large to all aspects in the management of information systems, and how to integrate the techniques applied in reuse-in-the-large into environments for information system development covering dynamic modelling.

Summary

Reusability is a central concept in life-cycle technology, in system evolution, and in the development of theories and models.

– Peter Wegner, 1984

Motivation. Many dreams of automation have become true since the application of information systems, but the dream to automate the development and maintenance of information systems is still a dream. Fortunately, application reuse is known as a promising technology for such automation, though many problems remain to be solved. This dissertation aims at seeking solutions for these problems and providing techniques for the automation.

The establishment of RITL (the information system and the approach for application reuse-in-the-large) is motivated by the observation of the representation problems which are reflected in two research fields of application reuse: forward engineering and reverse engineering. Whereas a lot of information fails to be represented in system development (forward engineering), quite similar information has to be recovered for the purpose of reuse (maintenance or re-engineering), i.e. system producers make puzzles and let reverse engineers solve them. The redundant work is against the basic principle of reuse. Unfortunately, upto recent years, no great attention has been paid to representation problems concerning application reuse. Webster's research on representing technologies shown that conventional systems do not adequately cover the representation needs of a design process. Conventional systems exhibit what could be called representational myopia. They are hard to be reused and have little or no semantic basis in many cases. Rugaber and his colleagues discovered that system design involves making choices among alternatives, but too often, however, the alternatives considered are lost in system

representation. They indicated that one reason of loosing design information is that the commonly used design representation is not expressive enough, the other reason is that the design representation fails to describe the process by which design decisions are reached.

The establishment of RITL is also motivated by observing the problems of reusing source code components. Shaw described that the knowledge expressed in the form of code can not be useful if programmers do not know about it or are not encouraged to use it, and the library components require more care in design, implementation and documentation than similar components that are simply embedded in systems. Furthermore, Biggerstaff indicated that building systems out of small (source code) components leaves a lot of work to be done. While the components are made larger and larger, they become less and less reusable. Therefore, he addressed the need to eliminate some of the specificity necessitated by a source code-oriented specification, and to seek representations that allow the large-grain component structure to be described precisely, while leaving many of the small, relatively unimportant details uncommitted.

The dissertation work can be largely divided into 4 parts: research setting, preliminary exploration, modelling and prototyping. During the research we encouraged ourselves to reuse all the achievements known by us. As such we could avoid reinventing wheels and focused on something new.

Research setting. For the research setting we classified the techniques for application reuse into reuse-in-the-large and reuse-in-the-small, aiming at identifying the requirements for system development and maintenance. Reuse-in-the-small reflects the needs of programming-in-the-small where algorithms and coding are especially important, e.g. making a program for sorting. Reuse-in-the-large reflects the needs of programming-in-the-large where analysis, design and maintenance become very important, e.g. building an information system. Therefore, our research was centered on the methodology for reuse-in-the-large, aiming at improving the process of application development and maintenance.

Understanding the state of the art in the field of reuse-in-the-large, we investigated many approaches in the field of reuse, including Neighbors' Draco approach for domain analysis, Havelund and Haxthausen's EPROS approach for formal specification and prototyping, Biggerstaff's Desire system for design recovery, Booch's approach for object-oriented design, and Basili's approach

for re-engineering, and several model interconnection languages for the description of system structure.

Although most of the approaches investigated are successful in reaching expected goals, we have found neither an approach which supports an integrated process for the creation and reapplication of system components, nor an approach which supports an integrated representation of an application, the information about the creation of the application, and the information about modification, reconstruction and management of the application in a structured manner.

Preliminary exploration. The preliminary exploration comprises our earlier work on application reuse including the research on the reuse of algorithm structures and abstract data types, and the research on the reuse of system architectures and design templates.

The research on reusing algorithm structures and abstract data types is a method for reuse-in-the-small. Such a method is based on two principles for component representations, i.e. object-oriented abstraction and structure-oriented abstraction. The former catches the commonality of operations which are able to operate on certain types of objects, which resembles what has been represented in object-oriented approaches; the latter catches the commonality of the operations whose implementations are isomorphic with respect to the algorithm structures used, which is (partially) what has been represented in some template- or frame-based approaches such as Bron's library of reusable modules.

The two principles lead to two kinds of components: object-oriented components and structure-oriented components. An object-oriented component is a component which represents an abstract data type or its extension, such as a class. A structure-oriented component is a component which represents a set of interrelated algorithm structures and a set of operations with respect to the algorithm structures.

The result of such research indicates that the context for component composition, as represented in terms of algorithm structure, may play an important role in reusing source code, and component modelling may not only be useful to package operations oriented to certain objects, but also to package the operations with respect to certain algorithm structures.

As another preliminary exploration, we engaged in a research on the reuse of system architectures and design templates, aiming at reusing the artifacts

which are produced during system design.

A system architecture is an organizational structure of a (application) system, which normally refers to the relationships between different parts of a system and can be mechanically recovered from source code. However, a system architecture in our approach is derived from the process of design decomposition, representing different levels of designs and the relationships between them. In terms of system architecture, one may understand an application easily and maintain an application at different levels of abstraction. The reusability of a system architecture implies the capability of supporting the maintenance, evolution and reconstruction of the architectures, resulting in varieties of design instances in an application domain.

A design template is a mapping from components to components, which can be represented in terms of a generic design module, i.e. a common frame of reference for a set of alternative designs in an application domain. The advantage of using design templates is the possibility of reducing the redundancy of similar design representations, which is useful for the modification and reconstruction of an application.

Modelling. Our research on reuse-in-the-large is centered on the representation, management and manipulation of large-scale components, resulting in RITL, an approach and an information system for application reuse-in-the-large.

RITL is a novel approach supporting application development and maintenance with reuse-in-the-large. The idea of reuse-in-the-large, in contrast with that of traditional reuse (the reuse of small source-code components), aims at reusing large-grain components and, more important, the information about the creation of the components.

This approach draws from research into re-engineering and forward engineering. From re-engineering research, we understood the necessity of reusing design information and the possibility of reconstructing a similar system from existing one. From forward engineering we discovered suitable structures to represent design information.

This approach is a multiple-level one, dealing with the *composition process* at each level of abstraction and within the context of (1) a *design framework*, acting as an algorithm explaining how to organize lower level entities into alternative design instances, (2) a collection of *design instances*, each describing a particular design, (3) *domain resources* consisting of a domain model and a

set of domain entities, and (4) a *refinement* as a detailed design and a particular implementation of a design instance. These four concepts are combined into a large-scale component, encapsulating both a component's design and the context in which the design is realized, by which both the design information and large-grain source code can be managed and manipulated to meet different design targets of an application domain.

According to this approach, application developers are encouraged to reuse design information as well as source code. At the same time, they are encouraged to provide information for maintenance. Moreover, application maintenance can be viewed as application development with reuse-in-the-large in an application domain. As such, forward engineering and re-engineering may follow the same process and use the same infrastructure — large-scale components — and the tools for understanding, and manipulation of the components. The advantage of using such an infrastructure is the possibility of incremental domain analysis and semi-automatic component retrieval.

This approach can be supported by an information system. The information representation of such a system is based on a pragmatic model, formalizing the information for application reuse-in-the-large, and an operating model, formalizing the activities for application development and maintenance with application reuse-in-the-large. The systematic process of application development and maintenance with reuse-in-the-large is potentially determined by these two models.

The information system is furnished in terms of specification languages, programming languages, a semantic data base, a user interface, and a set of tools to support the management and manipulation of large-scale components.

Moreover, in order to represent and understand large-scale components in a precise ways so as to reuse them effectively, we introduce a principle for the specification of large-scale components and describe how to apply such a principle for the representation of large-scale components. Besides, we address a method for the consistency verification of large-scale components. This method suggests a diagram guided semantic verification, by which a refinement verification is straightforward and efficient. It is straightforward as the verification can be done with semantically extended data flow graphs which represent system design and system analysis. It is efficient as such a verification can be simplified with a set of inference rules and guided with graphical structures which can be intuitively understood.

Prototyping. As an investigation of applying the approach for reuse-in-the-large to the practice of system development, we looked into how a large-scale component is capable of supporting system prototyping. By such an example the behaviour of reuse-in-the-large was investigated, including the representation of a large-scale component, the process of application prototyping with reuse-in-the-large, the activities for application reconstruction, and the possibility of automatic component retrieval.

In order to test how RITL is practically applied in more specific ways, we discussed how reuse-in-the-large achieved in a support environment, regarding a particular application domain: Student Management System (SMS), which is represented by a large-scale component. Particularly, we discussed how SMS is specified in terms of an available specification language, how it is understood and manipulated, how varieties of designs and implementations can be generated from the large-scale component to meet varieties of requirements in an application domain.

Conclusion. Consequently, this dissertation work provides system developers with a set of approaches for application reuse, particularly for application reuse-in-the-large. The approaches are concerned with (i) the reuse of abstract data types and algorithm structures, providing a means for the construction of source code components, (ii) the reuse of system architectures and design templates, providing a means for the representation of design information, and (iii) the reuse of large-scale components, providing an infrastructure for representation, management, manipulation and reapplication of both applications and the information about the creation of the applications. The approach for the reuse of large-scale components results in an information system, a support environment for application development and maintenance with reuse-in-the-large.

Appendix A

Scomps and Ocomps in Duplex Shell

Duplex shell is a software tool which supports programming with Scomps and Ocomps. In this appendix the semantics of the Duplex shell is described with a set of objects[1], axioms and transformation rules. Based on this description, some theorems are presented and an operation generation is specified. Additionally, the syntax of the system components is appended.

1. Predicates and notations

Conventions As a rule, an identifier beginning with a capital letter stands for a set of objects, whereas an identifier containing no capital letter stands for an instance of object. Another convention is that the form of the logical formula $\forall x \in M(predicate(x))$ is equivalent to $\forall x((x \in M) \wedge predicate(x))$.

Predicates Three predicates are defined in this appendix. The expressions with form $y =_{df} < x_1, x_2, ...x_n >$ in this appendix refer to the definition of an object. An object is a tuple. An element of the tuple may be: a language construct, another object or a set of other objects. See also section 3 for details. The three predicates are defined as follows:

[1]the notion *objects* in this appendix means some things to be dealt with. It is quite different from the objects in object-oriented programming.

- $element(x; y)$ is a predicate which may be read as x *is an element of* y. The definition is as follows:

  $\forall x, y, z \in (Object \cup Object')$

  $$element(x; y) = \begin{cases} true & \text{if } y =_{df} < ..., x, ... >; \\ true & \text{if } y \text{ is a set, and } x \in y; \\ true & \text{if } y =_{df} < ..., z, ... > \wedge x \in z, \ z \text{ is a set;} \\ false & \text{otherwise.} \end{cases}$$

  Set $Object$ contains all the objects that are dealt with in the Duplex shell. Set $Object'$ contains sets of the objects. For an exact definition see also section 3.3. As an abbreviation, $element(x_1, x_2; y)$ is equivalent to $element(x_1; y) \wedge element(x_2; y)$.

- $element'(x; y)$ is a predicate which may be read as x *is an element or sub-element of* y.

  $\forall x, y, z \in (Object \cup Object')$

  $$element'(x; y) = \begin{cases} true & \text{if } element(x; y); \\ true & \text{if } \exists z(element'(x; z) \wedge element'(z; y)); \\ false & \text{otherwise.} \end{cases}$$

  Set $Object$ and set $Object'$ are the same to the sets used for defining $element(x; y)$. The abbreviated form is also the same.

Notations Two notations are defined as follows:

- $name(x)$: is a notation which denotes the name of x;
  $x \in Object$.

- $signature(x)$: is a notation which denotes the signature of x;
  $x \in (Oprt \cup Voprt \cup Oprtspec)$. The sets $Oprt$, $Voprt$ and $Oprtspec$ are defined in section 3 and 4.

- $y[t/x]$: is a notation which denotes a new object created by substituting all occurrences of x in y by t; $x, y, t \in Object$ $\wedge$ $element(x; y)$.

2. Objects in programming languages

The Duplex shell is independent from the programming languages which might be used for programming in Duplex shell. However, the Duplex shell is designed, based on the following definitions.

$Datatype =_{df} \{x | x$ is a definition of data type and $\exists ! y \in Id(name(x) = y).\}$[2††]

$Udatatype =_{df} \{x | x \in Datatype$ which is defined by users.$\}$

$Sdatatype =_{df} \{x | x \in Datatype$ which is defined in programming language.$\}$

$Oprt =_{df} \{x | x$ is an implementation of a user-defined operation(program unit); and $\exists ! s \in Oprtsig(element(s; oprt) \wedge signature(x) = s \wedge name(oprt) = name(s)\ )\}$

$Expr =_{df} \{x | x$ is an expression in a programming language.$\}$

$Id =_{df} \{x | x$ is an identifier in a programming language.$\}$

$Oprtsig =_{df} \{x | x$ is the heading of an operation in a programming language; $\exists ! y \in Id(element(y; x) \wedge name(x) = y\ ).\}$

$Variable =_{df} \{x | x$ is a variable in a programming language.$\}$

3. Objects in the Duplex

3.1. Basic objects and object sets

The basic objects in the Duplex shell are defined as tuples with the form $x =_{df} < x_1, x_2, ..., x_n >$: *expression* which is read as " x consists of $x_1, x_2, ..., x_n$ with the condition of *expression*"; the *expression* is a Boolean expression.

[2††] We assume that any data type in programming language has a name. Although there might be anonymous data types, any anonymous data type can be substituted by a named data type, which is consistent with the application in the Duplex shell.

As a convention, each object defined in this section implies an object set. We use an identifier beginning with a capital letter to stand for the set, while the identifier in lower-case letters stands for an instance of the object. For example, the virtual operation is defined in the form $voprt =_{df} < ... >$ This definition implies a definition of a set: $Voprt =_{df} \{voprt | voprt =_{df} < ... >\}$.

In addition, the object sets used in the section, Cmp, and $Cname$ are defined in 4.3.

- **Duplex shell,** *duplex*, is a 3-tuple which consists of a set of Scomp, *Scp*, a set of Ocomp, *Ocp*, and an operation generator, *generator*.

$$duplex =_{df} < Scp, Ocp, generator > .$$

- **Ocomp,** *ocp* , is an 8-tuple which consists of the name of the Ocomp, *oname*, a set of data types, *Datatype'*, a set of public variable declarations, *Pubdata'*, a set of private variable declarations, *Pridata*, a set of Operation implementations, *Oprt'*, a set of virtual operations, *Voprt'*, and *Inherit*, which is a set of names of other Ocomps inherited by the Ocomp.

$$\begin{aligned} ocp =_{df} < oname, Datatype', Pubdata', Pridata', Oprt', Voprt', Inherit' >: \\ oname \in Id \wedge name(ocp) = oname \wedge \\ Datatype' \subset Datatype \wedge Pubdata' \subset Pubdata \wedge \\ Pridata' \subset Pridata \wedge Oprt' \subset Oprt \wedge \\ Voprt' \subset Voprt \wedge Inherit' \subset Inherit. \end{aligned}$$

- **private variable declaration,** *pridata*, is a 3-tuple which consists of a mark for privacy, *primark*, a name of a data type, *typename*, and a set of variables, *Privariable*.

$$\begin{aligned} pridata =_{df} < primark, typename, Privariable >: primark \in Id \wedge \\ typename = name(datatype) \wedge datatype \in Datatype \wedge \\ Privariable \subset Variable. \end{aligned}$$

- **public variable declaration,** *pubdata*, is a 3-tuple which consists of a mark for publicity, *pubmark*, a name of a data type, *typename*, a set of variable, *Pubvariable*.

$$pubdata =_{df} < pubmark, typename, Pubvariable >, pubmark \in Id \wedge \\ typename = name(datatype) \wedge datatype \in Datatype \wedge \\ Pubvariable \subset Variable.$$

- **virtual operation,** *voprt*, is a 2-tuple which consists of an operation signature, *oprtsig*, and a name of an Scomp, *sname*. The name of the operation signature is the name of the virtual operation.

$$voprt =_{df} < oprtsig, sname >: oprtsig \in Oprtsig \wedge \\ signature(voprt) = oprtsig \wedge name(voprt) = name(oprtsig) \wedge \\ sname \in ID \wedge \ \exists! scp \in Scp(sname = name(scp)).$$

- **inherit,** *inherit*, is a name of an Ocomp.

$$inherit =_{df} < x >: x = name(ocp) \wedge ocp \in Ocp.$$

- **Scomp,** *scp*, is a 6-tuple which consists of a name of the Scomp, *sname*, a set of algorithm structure specifications, *Structspec'*, a set of algorithm structures, *Struct'*, a set of operation specifications, *Oprtspec*, a default instantiation description, *default*, and a set of instantiation descriptions, *Desc'*.

$$scp =_{df} < sname, Structspec', Struct', Oprtspec', default, Desc' >: \\ sname \in Id \wedge name(scp) = sname \wedge \\ Structspec' \subset Structspec \wedge Structspec' \neq \phi \\ Struct' \subset Struct \wedge Struct' \neq \phi \\ Oprtspec' \subset Oprtspec \wedge default \in Default \wedge \\ Desc' \subset Desc.$$

- **algorithm structure specification,** *structspec*, is a 3-tuple which consists of the name of the algorithm structure, *structname*, a set of generic abstract data types, *Abstype'*, and a set of generic expressions, *Mexpr'*.

$$structspec =_{df} < structname, Abstype', Mexpr' >: \\ structname \in Id \wedge Abstype' \subset Abstype \wedge$$

$$Mexpr' \subset Mexpr \wedge name(structspec) = structname.$$

- **algorithm structure,** ***struct***, is an operation in which one or more names of data types, one or more names of operations, and/or one or more expressions are substituted by the generic parameter,*macro*. In addition, the name of the operation is the name of algorithm structure.

$$struct =_{df} < x >: x = oprt[macro/atom] \vee x = struct[macro'/atom'] : \\ oprt \in Oprt \wedge macro, macro' \in Macro \wedge (\exists y, y' \in \\ (Elem \cup Datatype)(atom = name(y), atom' = name(y')) \vee \\ (atom, atom' \in Expr)) \wedge name(struct) = name(oprt).$$

- **operation specification,** ***oprtspec***, is a 2-tuple which consists of an operation signature, *oprtsig*, and a set of component names,*Cname''*.

$$oprtspec =_{df} < oprtsig, Cname'' >: signature(oprtspec) = oprtsig \\ name(oprtspec) = name(oprtsig) \wedge Cname'' \subset Cname.$$

- **default instantiation description,** ***default***, is a 3-tuple which consists of a mark for default, *demark*, a set of generic assignments, *M'*, and a set of component names, *Cname'*.

$$default =_{df} < demark, M', Cname' >: demark \in Id \wedge M' \subset Massign \wedge \\ Cname' \subset Cname.$$

- **instantiation description,** ***desc***, is a 2-tuple which consists of the name of the instantiation description, *descname*, and a set of generic assignments, *M''*.

$$desc =_{df} < descname, M'' >: descname \in Id \wedge M'' \subset Massign \wedge \\ name(desc) = descname.$$

- **generic abstract data type,** ***abstype***, is a 2-tuple which consists of a set of generic data types, *Mdatatype'*, and a set of generic operations, *Moprt'*.

$$abstype =_{df} < Mdatatype', Moprt' >: \\ Mdatatype' \subset Mdatatype \wedge Moprt' \subset Moprt.$$

- **generic data type,** *mdatatype*, is a generic parameter, *macro* or a name of a user-defined datatype.

$$mdatatype =_{df} < x >: (x \in Macro) \vee \exists! y((y \in Sdatatype) \wedge \\ (x = name(y)).$$

- **generic operation,** *moprt*, is a 4-tuple which consists of the name (generic) of the operation, *mname*, a set of generic inputs, *Minput*, a set of generic outputs, *Moutput*, and a set of accessible generic data types, *Maccess*.

$$moprt =_{df} < mname, Minput, Moutput, Maccess >: \\ mname \in Macro \wedge Minput \subset Mdatatype \wedge \\ Moutput \subset Mdatatype \wedge Maccess \subset Mdatatype.$$

- **generic expression,** *mexpr*, is a 2-tuple which consists of the name (macro) of the expression, *exprname*, and a set of expressions, *Expr'*.

$$mexpr = < exprname, Expr' >: exprname \in Macro \wedge Expr' \subset Expr.$$

- **generic parameter,** *macro*, is an identifier which is an element of a macro assignment.

$$macro =_{df} < identifier >: identifier \in Id \wedge \exists massign \in Massign(\\ element(identifier; massign))$$

- **value,** *value*, is a name of an element of Elem (see also 4.3), a name of a data type, or an expression.

$$value =_{df} < v >: \exists x \in (Datatype \cup Elem)(v = name(x)) \\ \vee (v \in Expr).$$

- **generic assignment,** *massign*, is a 2-tuple which consists of an identifier (generic parameter) and a value.

$$massign =_{df} < identifier, value >: value \in Value \land \\ identifier \in Id.$$

- **full name,** $fullname(x)$, is a 2-tuple which consists of the name of the element x and the name of a component to which the element belongs.

$$fullname(x) =_{df} < name(x), name(cmp) >: element(x; cmp) \land cmp \in Cmp.$$

- **operation generator,** *generator*, is defined in a separate section "Operation generator".

$$generator =_{df} < ... >$$

3.2. Auxiliary definitions

In this section a complete instantiation description, *descx*, is defined in terms of an instantiation description, *desc* and its default, *default*.

Let $scp \in Scp$;
$element(desc, default, oprtspec; scp)$;
$element(descname, M''; desc)$;
$element(M', Cname'; default)$;
$element(Cname''; oprtspec) \land name(oprtspec) = name(desc)$;

$$descx =_{df} < descname, M''', Cname''' >: \\ Cname''' =_{df} \{x | x \in (Cname' \cup Cname'')\} \\ M''' =_{df} \{x | x \in M'' \lor (x \in M' \land \neg\exists macro \in Macro \\ (element'(macro; M'') \land element(macro; x))\}$$

3.3. Additional object sets

By using the definitions in section 3.1, 3.2, we define some other object sets as follows:

– the elements of an algorithm structure, the name of the elements.

$$Elem =_{df} \{x|x \in (Oprt \cup Oprtspec \cup Voprt \cup Struct \cup Udatatype)\}$$
$$ElemName =_{df} \{x|\exists y(y \in Elem \wedge x = name(y))\}$$

– The members of an Ocomp, the names of the members.

$$Memb =_{df} \{x|x \in (Oprt \cup Voprt \cup Pubdata \cup Datatype)\}$$
$$MembName =_{df} \{x|\exists y(y \in Memb \wedge x = name(y))\}$$

– Components, the name of the components.

$$Cmp =_{df} \{x|x \in (Scp \cup Ocp)\}$$
$$Cname =_{df} \{x|\ \exists cmp \in Cmp(x = name(cmp))\}$$

– The basic object sets.

$$\begin{aligned} Object' =_{df} \{&Cmp, Datatype, Expr, Oprt, Id, Oprtsig, Variable,\\ &Struct, Structspec, Oprtspec, Default, Desc, Descx,\\ &Massign, Value, Macro, Pridata, Pubdata,\\ &Voprt, Inherit\}\end{aligned}$$

– Basic objects, the names of the objects.

$$Obj =_{df} \{x|\exists y(x \in y \wedge y \in Object')\}$$
$$ObjName =_{df} \{x|\exists y(y \in Obj \wedge x = name(y))\}$$

– Objects.

$$Object =_{df} \{x|(x \in (Obj \cup ObjName)\}$$

4. Axioms and transformation rules

The notation $transform$ is defined as: *for all information x and y, of which x is said to be transformed into y, holds that there is a function (practical algorithm) with input x and output y.* For further discussion, three symbols are defined as follows:

- $x \longrightarrow y$ or $\frac{x}{y}$: x can be transformed into y. As an abbreviation, $x \longleftrightarrow y$ denotes $(x \longrightarrow *y) \wedge (y \longrightarrow *x) : x, y \in Object$; and $x \longrightarrow y, z$ denotes $(x \longrightarrow y) \wedge (x \longrightarrow z) : x, y, z \in Object$;
- $x \longrightarrow *y$: $(x \longrightarrow y) \vee \exists z \in Object((x \longrightarrow *z) \wedge (z \longrightarrow *y)) :$ $x, y \in Object.$
- $x \longmapsto y$: A map from domain x to domain y: $x, y \subset Object.$ To be sure not to mix up things, we also use $x \longleftrightarrow y$ to denote $x \longmapsto y \wedge y \longmapsto x.$
- **ObjName** $\longleftrightarrow$ **Object**;

$\forall scp \in Scp, ocp \in Ocp$
Let
$scp =< sname, Structspec', Struct', Oprtspec', default, Desc' >$
$ocp =< oname, Datatype', Pubdata', Pridata', Oprt', Voprt', Inherit' >$
$EE =_{df} \{Cmp, Structspec', Struct', Oprtspec', Desc', Datatype', Oprt', Voprt'\}$

  - Object names in certain domains must be unique.

$\forall E \in EE, x, y \in E$

$$name(x) = name(y) \longleftrightarrow x = y \quad (A.1)$$

  - A component name can be transformed into the component and vice versa.

$\forall cmp \in Cmp$

$$name(cmp) \longleftrightarrow cmp \quad (1.1)$$

- **Voprt $\longmapsto$ Oprtspec $\times$ Scp**

  $\forall voprt \in Voprt \; \exists! scp \in Scp, oprtspec \in Oprtspec.$
  Let $voprt =< oprtsig, sname >$

  - A virtual operation, *voprt*, can be transformed into an operation specification, *oprtspec* and an Scomp, *scp*

$$voprt \longrightarrow oprtspec, scp \tag{A.2}$$

  - If a virtual operation, *voprt*, is transformed into an operation specification, *oprtspec*, and an Scomp, *scp*, then the operation specification is an element of the Scomp, and the signature of the operation specification is the same to the signature of the virtual operation.

$$\frac{voprt \longrightarrow oprtspec, scp}{\begin{array}{c} name(scp) = sname \\ element(oprtspec; scp) \\ signature(oprtspec) = oprtsig \end{array}} \tag{A.3}$$

- **Inherit**:

  - If the name of an Ocomp, ocp', appears in an inherit list, *Inherit*, of another Ocomp, *ocp*, all members ($\in Memb$) of the Ocomp belong to the other Ocomp, except the elements which are redefined in the other Ocomp.

  let
  $ocp =< oname, Datatype', Pubdata', Pridata', Oprt', Voprt', Inherit' >$
  $\forall ocp, ocp' \in Ocp, Inherit' \subset Inherit$

$$\frac{\begin{array}{c} name(ocp') \in Inherit' \wedge \\ element(Inherit'; ocp) \end{array}}{\forall x', x \in Memb \left(\left(\begin{array}{l} element'(x'; ocp') \wedge \\ element'(x; ocp) \wedge \\ name(x') \neq name(x) \end{array} \right) \longrightarrow element(x'; ocp) \right)} \tag{A.4}$$

- **Variable**

  – In an Ocomp a variable can be private or public, but not both.

$\forall ddecl', ddecl'' \in (Pridata \cup Pubdata), variable' \in Variable,$
$ocp \in Ocp$

$$\frac{\begin{array}{c} ddecl' \in Pridata \\ ddecl'' \in Pubdata \\ element(Pridata, Pubdata; ocp) \end{array}}{element(variable'; ddecl') \longrightarrow \neg element(variable'; ddecl'')} \tag{A.5}$$

- **Oprtspec $\longleftrightarrow$ Desc**

  – In an Scomp an operation specification, *oprtspec*, can be transformed into an instantiation description, *desc*, and vice versa. The names of both are the same.

$\forall oprtspec \in Oprtspec, scp \in Scp, \exists! desc \in Desc$

$$\frac{element(oprtspec; scp)}{\begin{array}{c} element(desc; scp) \wedge \\ name(desc) = name(oprtspec) \end{array}} \tag{A.6}$$

$$\frac{element(desc; scp)}{\begin{array}{c} element(oprtspec; scp) \wedge \\ name(desc) = name(oprtspec) \end{array}} \tag{A.7}$$

- **Desc $\longmapsto$ Struct**

  – In an Scomp to any instantiation description, *desc*, there is one and only one algorithm structure, *struct*, in the Scomp, and one and only one generic assignment, *massign*, in the instantiation description so that the generic parameter, *macro*, of the generic assignment is identical with the name of the algorithm structure.

$\forall desc \in Desc, scp \in Scp$
Let $scp =< sname, Structspec', Struct', Oprtspec', default, Desc' >$
$desc =< descname, M'' > \wedge desc \in Desc'$

$$element(desc, Struct'; scp) \longrightarrow \begin{array}{l} \exists! massign \in M'' \\ \exists! struct \in Struct' \\ (massign =< name(struct), \\ \qquad descname >) \end{array} \tag{A.8}$$

- **Default × Desc ⟼ Descx**

  - In an Scomp an instantiation description, *desc*, and its *default*, can be transformed into a complete instantiation description, *descx*.

$\forall scp \in Scp, default \in Default, desc \in Desc, \exists! descx \in Descx$

$$\frac{element(default, desc; scp)}{default, desc \longrightarrow descx} \tag{A.9}$$

- **Descx × Struct ⟼ Oprt**

$\forall desc \in Desc, scp \in Scp$
Let $scp =< sname, Structspec', Struct', Oprtspec', default, Desc' >$

  - In an Scomp a complete instantiation description, *descx*, and an algorithm structure, *struct*, can be transformed into an operation, *oprt*.

$$\frac{\begin{array}{c} element(desc; Desc') \wedge \\ default, desc \longrightarrow descx \end{array}}{\begin{array}{c} \forall struct \in Struct', \exists oprt \in Oprt \\ (descx, struct \longrightarrow oprt) \end{array}} \tag{A.10}$$

  - For each instantiation description, *desc*, there is one and only one algorithm structure, *struct*, in an Scomp. If the instantiation description and its default

can be transformed into a complete instantiation description, the complete instantiation description and the algorithm structure can be transformed into an operation. As a result, the signatures of the operation are the same to those of an operation specification in the Scomp. The name of the operation specification is the same as the instantiation description.

$$\frac{\begin{array}{c} element(desc; Desc')\wedge \\ default, desc \longrightarrow descx \end{array}}{\begin{array}{c} \exists! struct \in Struct', oprt \in Oprt, oprtspec \in Oprtspec' \\ (descx, struct \longrightarrow oprt \wedge \\ name(desc) = name(oprtspec)\wedge \\ signature(desc) = signature(oprt) \end{array}} \qquad (A.11)$$

- If in an Scomp an operation, *oprt*, is transformed from a complete instantiation description, *descx*, and an algorithm structure, *struct*, then the operation is produced by substituting every generic parameter, *macro*, appearing in the algorithm structure with the corresponded value contained in the instantiation description.

$$\begin{array}{l} \forall struct \in Struct', descx \in Descx, oprt \in Oprt \\ \text{Let } descx =< descname, M''', Cname''' > \\ \qquad massign =< macro, value >: \forall massign \in M''' \end{array}$$

$$\frac{descx, struct \longrightarrow oprt}{\begin{array}{c} oprt = struct[value/macro]\wedge \\ element'(macro; struct) \end{array}} \qquad (A.12)$$

- **Struct $\longmapsto$ Struct**

- If the name of an algorithm structure, *struct*, appears in another algorithm structure, *struct'*, the two algorithm structures must belong to the same Scomp, scp.

$$\forall struct, struct' \in Struct$$

$$\frac{element(name(struct'); struct)}{\exists scp \in Scp(element(struct; scp)) \longrightarrow element(struct'; scp)} \qquad (A.13)$$

- **Macro $\longmapsto$ Value**

$\forall massign \in Massign, structspec \in Structspec.$
Let $structspec =< structname, Abstype', Mexpr' >,$
$El = Oprt \cup Voprt \cup Oprtspec \cup Struct,$
$Abstype' = \{abstype | abstype =_{df}< Mdatatype', Moprt' >\}$ and
$massign =< macro, value > .$

- If the type of a generic parameter is a generic data type, $macro \in Mdatatype'$, the value of the parameter must be a name of a data type.

$$macro \in Mdatatype' \longrightarrow \begin{array}{l}\exists datatype \in Datatype(\\ value = name(datatype))\end{array} \quad \text{(A.14)}$$

- If the type of a generic parameter is a generic operation, $macro \in Moprt'$, the value of the parameter must be a name of an element of El.

$$macro \in Moprt' \longrightarrow \exists elem \in El(value = name(elem)) \quad \text{(A.15)}$$

- If the type of a generic parameter is a generic expression, $macro \in Mexpr'$, the value of the parameter must be an expression, $expr$.

$$macro \in Mexpr' \longrightarrow \exists expr \in Expr(value = expr) \quad \text{(A.16)}$$

- **ElemName $\longrightarrow$ Elem**

$\forall scp \in Scp$
Let $scp =< sname, Structspec', Struct', Oprtspec', default, Desc' >$
$default =< demark, M', Cname' >$

$desc =< descname, M'' >$
$oprtspec =< oprtsig, Cname'' >$

– If in an Scomp a name of an element of *Elem* appears in a *default* instantiation description or in an algorithm *structure*, the element is an element of the Scomp or an element of another component. For the latter, the component name must be an element of *Cname'* in the default instantiation description.

$$\forall e \in Elem, struct \in Struct' \exists cmp \in Cmp$$

$$\frac{\begin{array}{c} element'(name(e); default) \vee \\ element'(name(e); struct) \end{array}}{\begin{array}{c} element'(e, cmp) \wedge \\ name(cmp) \in Cname' \\ \vee \\ e \in (Struct' \cup Oprtspec') \end{array}} \tag{A.17}$$

– If in an Scomp a name of an element of *Elem* appears in an instantiation description, *desc*, the element is an element of the Scomp or an element of another component. For the latter, the component name must be an element of *Cname''* in an operation specification, *oprtspec*. The name of the operation specification is the same to the instantiation description.

$$\forall e \in Elem, desc \in Desc', \exists oprtspec \in Oprtspec', cmp \in Cmp$$

$$\frac{element'(name(e); desc)}{\begin{array}{c} element(e, cmp) \wedge \\ name(desc) = name(oprtspec) \\ name(cmp) \in Cname'' \\ \vee \\ e \in (Struct' \cup Oprtspec') \end{array}} \tag{A.18}$$

- **Massign**

$\forall scp \in Scp$
Let $scp =< sname, Structspec', Struct', Oprtspec', default, Desc' >$
$default =< demark, M', Cname' >$

$$desc = < descname, M'' >$$

– In an instantiation description, *desc*, or its *default* of an Scomp, there are no two generic assignments, *massign* and *massign'*, containing the same generic parameter, *macro*.

$\forall macro \in Macro, massign, massign' \in M'$ (or $\in M''$)

$$\frac{element(macro; massign) \wedge \\ element(macro; massign')}{massign = massign'} \tag{A.19}$$

– If in an Scomp a generic parameter, *macro*, appears in an instantiation description, it must appear in the *default* instantiation description of the Scomp as well.

$\forall macro \in Macro$

$$element'(macro; M'') \longrightarrow element'(macro; M') \tag{A.20}$$

– In an instantiation description of an Scomp only one generic assignment, *massign*, is allowed to have a generic parameter, *macro*, which is identical with the name of an algorithm structure in the Scomp.

$\forall struct, struct' \in Struct', massign, massign' \in M'$
Let $massign = < macro, value >$
$massign' = < macro', value' >$

$$\frac{macro = name(struct) \wedge \\ macro' = name(struct') \wedge}{massign = massign'} \tag{A.21}$$

- **Macro**

– In an Scomp an identifier is a generic parameter, *macro*, of an algorithm structure, *struct*, if and only if the same identifier appears, as a generic parameter, in the default instantiation description of the Scomp.

$\forall id \in Id, macro \in Macro$

$$\frac{\begin{array}{c} element'(id; Struct') \wedge \\ element'(macro; M') \wedge \\ id = macro \end{array}}{element'(macro; Struct)} \tag{A.22}$$

- **Object $\longmapsto$ Object**

$\forall a, b \in Object$

$$\frac{element'(a; b)}{b \longrightarrow *a} \tag{A.23}$$

$$\frac{a \longrightarrow b}{a \longrightarrow *b}; \tag{A.24}$$

$$\frac{a \longrightarrow *b, b \longrightarrow *c}{a \longrightarrow *c} \tag{A.25}$$

$$\frac{a \wedge (a \longrightarrow *b)}{b} \tag{A.26}$$

5. Theorems

In this section several theorems are presented, which are proved in [70] based on the axioms and transformation rules defined in the previous section.

Theorem 1 *In the Duplex shell any virtual operation can be transformed into an operation through a mechanical method. The signature of the transformed operation is the same to the signature of the virtual operation.*

(a) $\forall voprt \in Voprt, \exists oprt \in Oprt$

$$voprt \longrightarrow *oprt \tag{A.27}$$

(b) $\forall voprt \in Voprt, \exists oprt \in Oprt$

$$\frac{voprt \longrightarrow *oprt}{signature(voprt) = signature(oprt)} \tag{A.28}$$

Theorem 2 *For all operation specifications and Scomps, if an operation specification is an element of an Scomp, the operation specification and the Scomp can be transformed into an operation; and if an operation specification and an Scomp are transformed into an operation, the signature of the operation is identical with the signature of the operation specification.*

$\forall oprtspec \in Oprtspec, scp \in Scp$

(a)

$$\frac{element(oprtspec; scp)}{oprtspec, scp \longrightarrow oprt} \tag{A.29}$$

(b)

$$\frac{oprtspec, scp \longrightarrow oprt}{signature(oprtspec) = signature(oprt)} \tag{A.30}$$

Theorem 3 *For all virtual operations, operations and any member, e, of the set Elem, there is a transformation. If a virtual operation is transformed into an operation following the transformation and the name of e occurs in the operation, then a set of component names can be transformed from the virtual operation, so that the name of the component to which the e belongs is a member of the set.*

$\forall e \in Elem, voprt \in Voprt, oprt \in Oprt, \exists \longrightarrow *$

$$\frac{voprt \longrightarrow *oprt \wedge element(name(e); oprt)}{\begin{array}{c}\exists Cname^a \subset Cname(voprt \longrightarrow *Cname^a) \wedge \\ \exists cmp \in Cmp(element'(e; cmp) \wedge (name(cmp) \in Cname^a)\end{array}} \tag{A.31}$$

Theorem 4 *For all virtual operations and operations, if a virtual operation is transformed into two operations, the two operations are identical with each other.*

$\forall voprt \in Voprt, oprt, oprt' \in Oprt, \exists \longrightarrow *$

$$\frac{voprt \longrightarrow *oprt', oprt''}{oprt' = oprt''} \tag{A.32}$$

Theorem 5 *The full name of an element of a component (Scomp or Ocomp) can be transformed into the element and the component.*

$\forall elem \in Elem, \exists! cmp \in Cmp$

$$fullname(elem) \longrightarrow elem, cmp : element(elem, cmp) \qquad (A.33)$$

6. Operation generator

The operation generator is a function which generates an implementation of any given virtual operation as well as all the implementations of its subroutines called by the operation and the data types referred to by the operation.

The algorithm of the operation generator is specified by using Pascal-like programming language constructs and the notations used previously in the semantic specification.

In the generator specification, the following assumptions are held:

$scp \in Scp; ocp, ocp', ocp'' \in Ocp; oprtspec \in Oprtspec;$
$struct \in Struct; voprt \in Voprt; oprt, oprt', oprt'' \in Oprt;$
$udatatype \in Udatatype; elem \in (Oprt \cup Udatatype)$

and the following predicate and function are available.

$atomic(elem)$:	predicate: true, if *elem* invokes no *oprt* other than itself; false, otherwise. *elem* is a program construct.
$get(x, y)$:	a function in which x is a notation, y is an object. To each call of the function, the function returns a different object which is denoted by x and is an element of y. It returns NIL if there is no more different object to be returned.

The specification of the operation generator is as follows.

```
Generator(voprt) : return set        /* ∃x((x ∈ Ocp) ∧ (element(voprt, x)) */
begin
      voprt ⟶ oprtspec, scp : element(oprtspec, scp);              / * A.1, A, 2 * /
      return(Generator'(oprtspec, scp)
end
```

```
Generator'(oprtspec, scp) : return set          / * element(oprtspec; scp) * /
 begin
       oprtspec, scp ⟶ oprt, Oprt';          /* theorem 2, A.6, A.10  */
        return (OprtOfScomp(oprt, Oprt', scp))
 end

OprtOfScomp(oprt, Oprt', scp) : return set   /*  oprt, Oprt' generated from scp */
 begin
Bag := {oprt} ∪ TypeOfScomp(oprt);
x := get(name(oprt'), oprt);
 while (x ≠ NIL )  do   /*  process the subroutines of oprt. */
 begin
        if ∃y((y ∈ Bag) ∧ (x = name(y))  then
                 ;
        else if ∃x, struct(x = name(struct) ∧ element(struct; scp))  then
                Bag := Bag ∪ OprtOfScomp(oprt', Oprt', scp);
                            /*  oprt' ∈ Oprt' ∧ name(oprt') = x, see also A.10 */
        else if ∃oprtspec(x = name(oprtspec) ∧ element(oprtspec, scp))  then
                Bag := Bag ∪ Generator'(oprtspec, scp);
        else if ∃voprt(x = name(voprt) ∧ element(fullname(voprt); oprt))  then
                 begin
                        fullname(voprt) ⟶ voprt, ocp;          /* theorem 5 */
                        Bag := Bag ∪ Generator(voprt)
                 end
        else if ∃oprt''(x = name(oprt'') ∧ element(fullname(oprt''; oprt)  then
                 begin
                        fullname(oprt'') ⟶ oprt'', ocp;          /* theorem 5 */
                        Bag := Bag ∪ OprtOfOcomp(oprt'', ocp)
                 end
       x := get(name(oprt'), oprt)
 end
 return(Bag)
 end
```

```
OprtOfOcomp(oprt, ocp) : return set   /* element(oprt; ocp) */
begin
        Bag := {oprt} ∪ TypeOfOcomp(, oprt, ocp);
        x := get(name(oprt'), oprt);
        while (x ≠ NIL) do   /* process the subroutines of oprt */
        begin
                if ∃y((y ∈ Bag) ∧ (x = name(y)) then /* recursive */
                        ;
                else if x = name(oprt') ∧ element(oprt'; ocp) then
                        Bag := Bag ∪ OprtOfOcomp(oprt', ocp);
                else if x = name(voprt) ∧ element(voprt; ocp) then
                        Bag := Bag ∪ Generator(voprt);
                x := get(name(oprt'), oprt)
        end
        return(Bag)
end

TypeOfScomp(oprt) : return set   /* oprt generated from an Scomp */
begin
        Bag := φ;
        x := get(fullname(udatatype), oprt);
        while (x ≠ NIL) do   /* process data types defined in ocp's */
          begin
                fullname(udatatype) ⟶ udatatype, ocp;        /* theorem 5 */
                if atomic(udatatype) then
                        Bag := Bag ∪ {udatatype};
                if ¬atomic(udatatype) then
                        Bag := Bag ∪ {udatatype} ∪
                                        TypeOfOcomp(udatatype, ocp);
                x := get(fullname(udatatype), oprt)
          end
        x := get(name(ocp'); oprt);
        while (x ≠ NIL) do   /* process data types which are ocp's */
        begin
                Bag := Bag ∪ {ocp'} ∪ inherit(ocp');
                x := get(name(ocp'), oprt)
        end
        return (Bag)
end
```

```
TypeOfOcomp(elem, ocp) : return set /* element(elem; ocp) */
 begin
       Bag := {elem};
       x := get(name(udatetype), elem);
        while (x ≠ NIL) do /* process udatatype defined in ocp */
        begin
               if element(fullname(udatatype); ocp) then /* inherit */
                      ;
               if ∃y((y ∈ Bag) ∧ (x = name(y)) then /* recursive */
                      ;
               else if atomic(udatatype) then
                      Bag := Bag ∪ {udatatype};
               else if ¬atomic(udatatype) then
                      Bag := Bag ∪ {udatatype} ∪
                                    TypeOfOcomp(udatatype);
              x := get(name(udatetype), elem)
        end
       x := get(fullname(udatatype), oprt);
        while (x ≠ NIL) do /* process inherited udatatype */
          begin
              fullname(udatatype) ⟶ udatatype, ocp;        /* theorem 5 */
               if atomic(udatatype) then
                      Bag := Bag ∪ {udatatype};
               if ¬atomic(udatatype) then
                      Bag := Bag ∪ {udatatype} ∪
                                    TypeOfOcomp(udatatype, ocp);
              x:=get(fullname(udatatype), oprt)
          end
       x := get(name(ocp'), oprt);
        while (x ≠ NIL) do /* process data types which are ocp's */
        begin
              Bag := Bag ∪ inherit(ocp');
              x := get(name(ocp'), oprt)
          end
        return(Bag)
 end
```

```
Inherit(ocp): return set  /* get inherited ocp's */
 begin
        Bag = {ocp};
        x := (get(name(inherit), ocp);  /* name(inherit) listed in the heading of ocp */
         while  (x ≠ NIL)  do
         begin
                Bag := Bag ∪ {ocp} ∪ inherit(ocp);
                x := (get(name(inherit), ocp)
         end
 end
         return (Bag)
 end
```

7. The syntax of the components

The syntax of the components (Scomps and Ocomps) is represented in an Extended Backs-Naur Form at an abstract level; the detailed language constructs are illustrated in natural language.

The syntax of the Ocomp specification

<Ocomp>::=‘**Ocomp**’ <Oname> [‘**inherit:**’] <inherit-list>
‘{’<data-part><operation-part>‘}’

<Oname>::=<identifier>
<inherit-list>::=[] | <Oname>{‘,’<Oname>}$_0^n$
<data-part>::=‘**data:**’ {<variable-decl>}$_0^n$
<operation-part>::=‘**operation:**’{<Ocomp-oper-spec>‘;’}$_0^n$

< identifier >::=< *defined in programming language* >
<variable-decl>::=<type><variable-list>
<Ocomp-oper-spec>::={ <Oper-signature>[:<Sname>]}$_0^n$

<type>::=<identifier>
<variable-list>::=<variable>{‘,’<variable>}$_0^n$
<oper-signature>::=< *a heading of a program unit*>
<Sname>::=<identifier>

<variable>::=<identifier>

The syntax of the Ocomp implementation

<Obody>::=**'Obody'** <Oname>
'{'<type-var-dd><operation-define>'}'

<type-var-dd>::=**'data:'**<type-define><private-variable-decl>
<operation-define>::=**'oper:'**<*a program unit*>

<type-define>::=<...>
<private-variable-decl>::=<variable-decl>

< variable-decl >::=< *variable declaration* >

The syntax of the Scomp specification

<Scomp>::=**'Scomp'**<Sname> '{' <Alg-spec-part><Oper-spec-part>'}'

<Sname>::=<identifier>
<Alg-spec-part>::=**'algorithm:'** {<alg-spec>';'$\}_1^n$
<Oper-spec-part>::=**'operation:'**{<Scomp-oper-spec>';'$\}_0^n$

<alg-spec>::=<stru-name>'{' { <M-abst-type> $\}_0^n$
{ <M-expr-list> $\}_0^1$ '}'

<Scomp-oper-spec>::=<signature>:<comp-list>

< stru-name >::=< identifier >
<M-abst-type>::='type:' { <M-type-list>';' $\}_0^1$
'oprt:' { <M-oprt-list> $\}_0^1$
<M-expr-list>::='expr:' { <M-expr-name>'={' <expr-list> '}' $\}_0^n$
<signature>::=<*a heading of a program unit* >
<comp-list>::={ [<Oname> | <Sname>] $\}_0^n$

<M-type-list>::=<M-type>{ ','<M-type>$\}_0^n$
<M-oprt-list>::=<M-oprt>';' { < M-oprt >';' $\}_0^n$
<M-expr-name>::=<identifier>
<expr-list>::=<expression>{ ','<expression> $\}_0^n$
<Oname>::=<identifier>
<Sname>::=<identifier>

<M-type>::=<identifier>
<M-oprt>::= <type><M-oprt-name>'('<type-list>')'<read-write>
< expression >::=< *defined in programming language* >

<type>::=<M-type> | <basic-type>
<M-oprt-name>::=<identifier>
<type-list>::=<type>{ ','<type>}$_0^n$
<read-write>::={ <RW> <type-list>';' }$_0^3$

<M-type>::=<identifier>
<basic-type>::=<identifier>
<type>::=<identifier>
<RW>::='RW' | 'R' | 'W'

The syntax of the Scomp implementation

<Sbody>::='**Sbody**'<Sname> '{' <Alg-desc-part><Oper-desc-part>'}'

<Sname>::=<identifier>
<Alg-desc-part>::='**algorithm:**' {<Algorithm-structure>';'}$_1^n$
<Oper-desc-part>::='**instant:**'{<inst-description>';'}$_1^n$

<algorithm-structure>::=<macro><text><algorithm-structure> |
<text><macro><algorithm-structure>
<inst-description>::=<desc-name>':'<assignment-list> |
'default:'<assignment-list>':'<comp-list>

< macro >::=< identifier >
<text>::=<*any original part of a program unit*>
<desc-name>::=<identifier>
<assignment-list>::=<assign> | <assign><assignment-list>
<comp-list>::={ [<Oname> | <Sname>] }$_0^n$

<assign>::='{'<macro>='{'<text>'}"}' |'{"* '<desc-name>'}'
<Sname>::=<identifier>
<Oname>::=<identifier>

Abbreviations

ABT, Abstype, abstype:	abstract data type.
Abody:	actor body.
BOOL:	Boolean.
Cmpt, Cmp, cmp:	component, Scomp or Ocomp.
Cname, cname:	component name.
Datatype, datatype:	data type.
Desc, desc:	instantiation description.
Descx, descx:	the instantiation description, merging default.
EDFG, edfg:	Extended Data Flow Graph.
EDFG:	Extended Data flow graph.
ELEM, Elem, elem:	element.
Expr,expr:	expression.
INST, inst:	instance.
LSC:	Large-Scale Component.
MIL:	Module Interconnection Language.
M:	macro.
Maccess:	macro of accessed data type.
Massign, massign:	macro assignment.
Mdatatype, mdatatype:	macro data type.
Mexpr, mexpr:	macro expression.
Minput:	macro input.
Moprt, moprt:	macro operation.
Moutput:	macro output.
OFS:	Output Fire Set.
Obase:	Object-oriented component base.
Obj:	object.

Obody:	Ocomp Body.
Ocomp, Ocp, ocp:	object-oriented component.
Oname, oname:	name of Ocomp.
Oprt, oprt:	operation.
Oprtsig, oprtsig:	operation signature.
Oprtsig, oprtsig:	operation signature.
Oprtspec, oprtspec:	operation specification.
POST, post:	Post Condition.
PRE, pre:	Precondition.
Pridata, pridata:	private data type declaration.
Privariable:	private variable.
Pubdata, pubdata:	public data type declaration.
Pubvariable:	public variable.
RITL:	Reuse-In-The-Large.
RMM:	Resource Management Menu.
RSL:	Raise Specification Language.
SEPDS:	Support Environment for Prototyping Distributed Systems.
SMS, sms:	Students Management System.
Sbase:	Structure-based component base.
Scomp, Scp, scp:	structure-oriented component.
Sdatatype:	system defined data type.
Sname, sname:	the name of Scomp.
Structspec, structspec:	the specification of algorithm structure.
Udatatype:	user-defined datatype.
Voprt, voprt:	virtual operation.
ddecl:	data declaration.
descname:	the name of an instantiation description.
exprname:	the name of expression.
fullname:	full name.
mname:	name of macro.
primark:	the mark of private data.
pubmark:	the mark of public data.
struct:	algorithm structure.
structname:	the name of algorithm structure.

Index

A
abstract algorithm, 19
abstract data type, 17
abstraction, 15
action, 39
active version, 169
actor, 38, 40, 105
adaptability, 18
aggregation, 58
algorithm, 19
algorithm structure, 17
algorithm tailoring, 19
alternative design, 166
analysis, 3, 79
application, 2
application domain, 11
application management, 176
application prototyping, 119
application reuse, 2
architecture, 5
artifact, 2
assertion, 69

B
browsing, 157

C
class, 19, 140
complete design instance, 79
component, 2
component classification, 5
component management, 4
component manager, 23
component repository, 5
component representation, 4
component retrieval, 5
composibility, 18
composition based approach, 18
composition process, 78
conceptual model, 69
connection consistency, 107
convertability, 70
current design, 169
current version, 169

D
data transform, 105
decomposition, 58
Def. 1 (*non-primitive actor*), 40
Def. 10 *(stronger constraints)*, 121
Def. 11 *(the refinement of an EDFG)*, 121
Def. 12 *(prototype)*, 122
Def. 2 (*generic link type*), 50
Def. 3 *(refinement consistency)*, 89
Def. 4 *(syntactic consistency)*, 106
Def. 5 *(connection consistency)*, 107

Def. 6 *(the refinement consistency of an actor)*, 107
Def. 7 *(well-formed edfg)*, 109
Def. 8 *(well-structured edfgs)*, 114
Def. 9 *(loosely constrained EDFG and actor)*, 120
design, 3, 80
design framework, 14, 60
design information, 2, 176
design instance, 14, 60
design process, 38
design template, 37
different levels of abstraction, 15
domain analysis, 81
domain entities, 60
domain instance, 96
domain model, 60
domain resources, 14, 60
dynamic modelling, 11

E
EDFG, 38, 44
edfg, 103
edfg-merging, 103
evolutionary prototyping, 119
external link, 104

F
figure 2.1. A reuse supp..., 24
figure 2.2. The syntax of ..., 27
figure 2.3. The syntax of ..., 29
figure 2.4. An Scomp Spec..., 33
figure 2.5. An Scomp Impl..., 34
figure 2.6. An Ocomp Spec..., 35
figure 2.7. An Ocomp Impl..., 35
figure 3.1. The definition ..., 39
figure 3.10. A template with ..., 55
figure 3.2 The module stru..., 39
figure 3.3. The represent..., 41
figure 3.4. The rules for ..., 43
figure 3.5. An actor, 45
figure 3.6. edfg and EDFG, 46
figure 3.7. The system ..., 47
figure 3.8. A rewritten ..., 51
figure 3.8. The state ..., 177
figure 3.9. A template, 52
figure 4.1. The abstractions..., 58
figure 4.2. The pragmatic ..., 62
figure 4.3. The operating ..., 64
figure 4.4. The specification ..., 66
figure 4.6.a. User interface ..., 74
figure 4.6.b. User interface ..., 74
figure 4.7. The process ..., 77
figure 4.8. An information ..., 84
figure 5.1. Hierarchical..., 90
figure 5.2. The refinement ..., 95
figure 5.3. Data proc..., 97–99
figure 5.3a. A type tree, 100
figure 5.4. An *edfg* and ..., 106
figure 5.5. The constructors ..., 112
figure 5.6. A (well-formed) ..., 114
figure 5.7. The constructors ..., 115
figure 5.8. A (well-str..., 118
figure 6.1. An example ..., 125
figure 6.2. An example of ..., 125
figure 6.3. An example ..., 126
figure 6.4. The refine..., 126
figure 6.5. The process ..., 127
figure 6.6. A large-scale..., 130
figure 7.1. The H-structure ..., 136
figure 7.2. The top-node of ..., 138

figure 7.3. Some outputs of ..., 139
figure 7.4. The Design..., 144
figure 7.5. The Design ..., 146
figure 7.6. The design..., 149
figure 7.7. The design ..., 151
figure 7.8. The Design ..., 154
figure 7.9. The design..., 156
figure I1. The user interface..., 158
figure I2. The user interface..., 160
figure I3. The user interface..., 162
figure I4. The user interface..., 167
figure I5. The user interface..., 168
flexibility, 18, 48
foreseen reuse, 30
forward engineering, 7
frame, 19
functional management, 176

G
generalization, 59
generic link types, 50
generic parameter, 131
genericity, 19
graphical actor, 104
graphical edfg, 104

H
hierarchical specification, 87
horizontal hierarchy, 40

I
I/O consistency, 107
IFS, 39
implementation, 3, 169
implementation generator, 83
import, 49, 53
incremental domain analysis, 13, 81
information hiding, 19
information system, 2, 176
inheritance, 5, 19, 49, 55
input link, 104
instance of a conceptual model, 69
instantiation, 59
interconnection between actors, 105
interim instance, 63
internal link, 104
iterative waterfall model, 12

L
large-grain component, 5
large-scale component, 14, 60
learning, 167
life-cycle model, 12
link, 39
link type, 40
loosely constrained actor, 120
loosely constrained EDFG, 120

M
management of information systems, 176
manipulating, 163
model, 12, 140
module, 17, 40
module structure, 40
multiple level specification, 87

N
non-primitive actor, 40
non-terminal nodes, 61

O
object, 18, 140
object-oriented abstraction, 22

objectivity, 70
Ocomp, 22
OFS, 39
operating model, 63
operation, 23
operation generator, 23
organizational structure, 63
output link, 104

P
parameterization, 59
parameterized class, 140
partial specification, 88
pragmatic model, 61
primitive actor, 40
primitive prototype, 128
process of system development and maintenance, 12
programming-in-the-small, 8
programming-in-the-large, 9
prototype, 122

Q
quality requirement, 18
queued link, 104

R
re-engineering, 7
refinement, 14, 61
refinement consistency, 88, 89
refinement of an EDFG, 38, 121
refinement structure of a large-scale component, 73
relatability, 70
retrieval problem, 69
reusability, 38
reuse, 2
reuse infrastructure, 11
reuse-in-the-large, 2
reuse-in-the-small, 2
reuse-supported forward engineering, 7
reverse engineering, 6

S
scheme, 140
Scomp, 22
semantic data modelling, 69
semantic data models, 69
Semantic databases, 69
simplicity, 18
small system components, 4
specialization, 59
state bearing object, 105
state link, 39, 104
stronger constraints, 121
structure-oriented abstraction, 22
syntactic consistency, 106
system architecture, 37
system component, 2
system design, 37
system development, 3
system life-cycle, 12
system maintenance, 3

T
tailoring, 19
technical management, 176
template, 19, 49
terminal nodes, 61
textual actor, 104
the state model, 176
the state of acceptance, 177

the state of development, 177
the state of exploitation, 177
the state of maintenance, 177
the state of use, 177
token, 39
tracing, 48
type, 69
type algebra, 69

U

unforeseen reuse, 30
union link type, 51
use-as-is, 22

V

version control, 169
vertical hierarchy, 40
virtual operation, 26

W

waterfall model, 12
well-formed edfgs, 109
well-structured edfgs, 114

Bibliography

[1] M. Abadi and L. Lamport, The existence of refinement mappings, *Proceeding of the Logic in Computer Science Conference*, Edinburgh, Scotland, July 1988.

[2] K. Abramowicz, K. Dittrich and others, Damoklas, *Data Base Management System for Design Application, reference manual*, Universitat Karlsruhe, release 2.0, March 1988.

[3] G. Arango, DOMAIN ANALYSIS – from Art To Engineering Discipline–, *Communication of ACM,* 1989 ACM 0-89791-305-1/89/05000/0152.

[4] C. W. Bachman, The Programmer as Navigator, *Comm. ACM,* Nov. 1993.

[5] R. Back, A Calculus of Refinements for Program Derivations, *Acta Informatica.* Vol.25, Springer-Verlag, 593-624.

[6] R. Back, K. Sere, Stepwise Refinement of Action Systems, *Structuring Programming*, 12:17-30, 1991.

[7] D. M. Balda, Cost Estimation Models for the Reuse and Prototype Software Development Life-Cycle, *ACM SIGSOFT, Software Engineering Notes*, vol. 15, No. 3, 1990.

[8] D. Barstow, Domain Specific Automatic Programming, *IEEE Transactions on Software Engineering*, vol. 7, Jan., 1985.

[9] V. R. Basili, Viewing Maintenance as Reuse-Oriented Software Development, *IEEE Software*, Jan. 1990.

[10] P. G. Bassett, Frame-based Software Engineering IEEE Software, July, 1987, pp. 9-19.

[11] J. H. ter Bekke, OTO-D: Object type oriented data modelling, Delft University of Technology, 1990.

[12] J. H. ter Bekke, *Semantic Data Modelling in Relational Environment,* PhD thesis, Delft University, The Netherlands, 1991.

[13] J. H. ter Bekke, *Semantic Data Modelling*, Print-Hall, 1992.

[14] V. Berzins and Luqi, *Software Engineering with Abstractions*, Addison Wesley Publishing Company, 1990.

[15] T. J. Biggerstaff and C. Richter, Reusability Framework, Assessment, and Directions, *IEEE Software* Vol 4 (2), March 1987, pp. 41-49.

[16] T. J. Biggerstaff, Reuse of very Large Scale Components, *MCC Tech. Report*, No. STP-363-88.

[17] T. J. Biggerstaff, Design Recovery for Maintenance and Reuse, *Computer*, July, 1989, pp 36-49.

[18] T. J. Biggerstaff and A. J. Perlis, *Software Reusability* Vol. I, II, ACM press, Frontier Series, 1989.

[19] G. Booch, *Software components with Ada Structures, tools, and subsystems,* The Benjamin/Cummings Publishing Company, 1987, pp. 21-22.

[20] G.
Booch, *Object Oriented Design with applications*, Benjamin/Cummings Publishing Company, 1991.

[21] M. F. Bott and P. J. L. Wallis, Ada and Software Reuse, *Ada Europe Software Reuse Seminar*, June, 1988.

[22] M. F.Bott, A. Elliott, and R. J. Gautier, Ada Reuse Guidelines, *Ada Europe Software Reuse Working Group*, March, 1987.

[23] B. A. Burton, R. W. Aragon and others, The Reusable Software Library, *IEEE Software* V 4 (4), 25-33, July 1987.

[24] E. W. Dijkstra, Programming Considered as a Human Activity, *Classics in Software Engineering,* New York, NY: Yourdon Press. p.5.

[25] R. C. J. Dur and P. G. W. Bots, Dynamic Modelling of organizations Using Task/Actor Simulation, in R. L. Croosslin and H. G. Sol (eds.), *Proceedings of the Second International Working Conference on Dynamic Modelling of Information Systems,* Elsevier Science Publishers (North-Holland), Amsterdam, 1991.

[26] C. Bron, Towards Libraries of reusable Module, *Computer Science notes,* Groningen University, The Netherlands, Jan. 1989.

[27] P. D. Bruza, Th.P.van der Weide, The Semantics of Data Flow Diagrams. Technical Report 89-16, University of Nijmegen, The Netherlands, Oct,. 1989.

[28] J. K. Chaar, 'Software Design Methodologies, A survey,' Technical Report RSD-TR-20-87, The University of Michigan, 1987.

[29] T. E. Cheatham, Reusability through Program transformations, IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984.

[30] A. Elliott, R. J. Gautier, and P. H. Welth, Component Engineering in Ada (some problem and some advise), *Ada Europe Software Reuse Seminar,* June, 1988

[31] J. Elliot, J.M. Chikofsky, Cross II, Reverse Engineering and Design Recovery: A Taxonomy, IEEE Software, Jan, 1990.

[32] J. H. Cross II, Reverse Engineering News Letter, Software Engineering Technical Committee News Letter, Jan, 1993.

[33] S. Conte, H. Dunsmore and others, A Software Metrics Survey, Rpt num: STP-284-86, August 20, 1986

[34] S. Cook, Languages and object-oriented programming, *Software Engineering Journal,* March 1986

[35] B. J. Cox, Planning the Software Industrial Revolution, *IEEE Software,* Sept. 1990.

[36] G. Cruman, Process programming: A Seductive Danger?, IEEE Software, May 1987, pp.91-93.

[37] M. Dewey, *Decimal Classification and Relative Index.* 19th ed. Albany, N.Y.: Forest Press, Inc., 1979.

[38] H. Fischer and G. D. Rozenblat, Reverse Engineering Speeds Software Development, *Electronic Engineering Times*, Issue 539, May 1989.

[39] M. K. Franklin and A. Gabrielian, A transformational Method for Verifying Safety Properties, *Real-time Systems Symposium* Santa Monica, California, 1989.

[40] R. B. France, Semantically Extended Data Flow Diagrams: A Formal Specification Tool, *IEEE Trans. on Soft. Eng.* vol.18, No. 4, Apr. 1992

[41] W. L. Frank, What Limits to Software Gains? *Computer world*, pp. 65-70, May 4, 1981.

[42] M. D. Fraser and others, Informal and Formal Requirements Specification Languages:Bridging the Gap, *IEEE Trans. on Soft. Eng.*, vol. 17, No. 5, May, 1991

[43] P. Freeman, Reusable Software Engineering: Concepts and Research Direction, in *workshop on Reusability in programming*, Alan Perlis, ed pp. 2-16, ITT Programming, Newport RI, Sept. 1983.

[44] A. Gabrielian and M. Franklin, Multiple-level Specification of real-time systems, *Communication of ACM*, Vol. 34, No. 5, May 1991.

[45] J.A. Goguen, J.W. Thatcher, and E. G. Wagner, An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, in Yeh, R. (ed). Current Trends in Programming Methodology, Prentice Hall, 1977, pp. 80-149.

[46] J. A. Goguen, Reusing and Interconnecting Software Components, *Computer*, Feb., 1986.

[47] K. Havelund and A. Haxthausen, *RSL, Reference Manual*, Pre-released document for VDM'90 RAISE tutorial, 1990.

[48] S. Hekmatpour and D. Ince, *Software Prototyping, Formal Methods and VDM*, Addison-Wesley Publishing Company, 1988.

[49] C.A.R. Haore and N. Wirth, An Axiomatic Definition of the Programming Language Pascal, Acta Informatica, Springer-Verlag Berlin, 1973. pp 335-355.

[50] E. Horowitz, and J. B. Munson, An Expensive View of Software Reusability, IEEE transactions on Software Engineering, SE-10, 5, Sept, 1984, 477-487.

[51] American National Standard Institute, IEEE standard Glossary of Software Engineering Terminology, An American National Standard, ANSI/IEEE, Std 729-1983 Aug., 1983.

[52] A. T. Jazaa and O. P. Brereton, Software Reuse in Ada, Computer Science Department, University of Keele, U.K., 1990.

[53] T. C. Jones, Reusability in Programming: A Survey of the State of the Art. *IEEE Trans. on Software Engineering,* V 10 (5), 488-493, Sept 1984.

[54] C. B. Jones, *System Software Development Using VDM*, Prentice/Hall international, Series in Computer Sciences, 1986.

[55] G. E. Kaiser, D. Garlan, Melding Software Systems from Reusable Building Blocks. IEEE Software V4 (4), July 1987, pp. 17-24.

[56] G. E. Kaiser, P. F. Feiler, and S. Popovich, Intelligence Assistance for Software Reuse and maintenance, *IEEE Software,* May, 1988.

[57] K. C. Kang, and L. S. Levy, *Software Reuse: What's Behind the Buzzword*, Technical Report cmu/SEI-89-TR-27, Software Engineering Institute, Carnegie Mellon University, May, 1989.

[58] J. van Katwijk, A. M. Levy, and others, Software Design for Distributed (Real-Time) systems using EDFG approach, Report, TWI, Delft Univ., Netherlands, 1990.

[59] D. E. Knuth, Fundamental Algorithms, *The Art of Computer Programming,* 2nd, ed., Vol.1, Addison-Wesley, 1973.

[60] K. Lano and P. T. Breuer, From Programs to Z Specification, Z User's Meeting, Dec. 1989.

[61] P. Larsen, N. Plat, and Hans Toetenel, A Formal Semantics of Data Flow Diagrams, submitted for a publication to Formal Aspects of Computing.

[62] L. Latour and others, *Theoretical Foundations Working Group Report*, Fourth Annual Workshop on Software Reuse, Reston, Virginia, 18-22, 1991.

[63] M. Lenz, A. Schmid, and P. F. Wolf, Software Reuse through Building Blocks, *IEEE Software,* July, 1987.

[64] P. Levy, and K. Ripken, Experience in Constructing Ada Programs from Non-Trivial Reuse Models, The Ada companion Series: Ada components and tools, Cambridge University Press, May 1987.

[65] A. M. Levy, H. Corporaal and J. van Katwijk, *Design of Parallel Resource Controller Using EDFG*, Parallel Computing 89, edited by D. J. Evans, G. R. Joubert and F. J. Peters, Elsevier Science Publishers B. V. (North Holland), 1990.

[66] A. M. Levy, J. van Katwijk, G. Pavlides, and Tolsma, *SEPDS: A Support Environment for prototyping distributed systems*, In Proceedings of the first International Conference on System Integration, New Jersey, USA, April 1990.

[67] H. Li, *An Introduction to Software Reuse*, Technical Report No. 91-50, ISSN 0922-5641, Faculty of Mathematics and Computer Science. Technical University of Delft, The Netherlands, 1991.

[68] H. Li, and J. van Katwijk, *Abstract Data types and algorithm structures as a basis for Software Reuse*, Technical Report 91-51, ISSN 0922-5641, Faculty of Mathematics and Computer Science. Technical University of Delft, The Netherlands, 1991.

[69] H. Li, J. van Katwijk, and J. van Zeijl, *Component Construction: Practical Guidelines for the Construction of Reusable Components*, Technical Report 91-89, ISSN 0922-5641, Faculty of Mathematics and Computer Science. Technical University of Delft, The Netherlands, 1991.

[70] H. Li and J. van Katwijk, *The Duplex Programming Environment*, Technical Report 91-90, ISSN 0922-5641, Faculty of Mathematics and Computer Science. Technical University of Delft, The Netherlands, 1991.

[71] H. Li and J. van Katwijk, *The Semantics of the Duplex Programming Environment*, Technical Report 91-91, ISSN 0922-5641, Faculty of Mathematics and Computer Science. Technical University of Delft, The Netherlands, 1991.

[72] H. Li and J. van Katwijk, A Model For Reuse-In-The-Large, *Proceeding of the 4th Annual Workshop on Software Reuse (WISR'91)* sponsored by the IEEE Computer Society, Reston, Nov. 18-22, 1991.

[73] H. Li and J. van Katwijk, Issues Concerning Software Reuse-in-the-Large, *Proceeding of the 2nd International Conference on Software Integration (ICSI)*, IEEE Computer Society Press, June, 1992.

[74] H. Li, J. van Katwijk and M. A. Levy, The Reuse of Software Design and Software Architecture, *Proceeding of the 4th International Conference on Software Engineering and Knowledge Engineering (SEKE'92)*, IEEE Computer Society Press, June, 1992.

[75] H. Li and J. van Katwijk, Reuse-in-the-Large for Software Prototyping, *Proceeding of the Fifth Nordic Workshop on Programming Language (NORDIC'92)*, Tampere, Finland, Jan. 1992.

[76] H. Li, The Reuse of Large-Scale Component Technical Report 92-95, ISSN 0922-5641, Faculty of Mathematics and Computer Science, Technical University of Delft, The Netherlands, 1992.

[77] H. Li, Reuse-in-the-Large: Modelling, Specification and Management, *Proceeding of the Second International Workshop on Software Reuse (IWSR'93)*, IEEE Computer Society, 1993.

[78] H. Li, Application Development and Maintenance with Reuse-in-the-large, IEEE International Conference on Computers, Communucation and Automation (TENCON'93), Beijing, Oct. 19-21, 1993.

[79] S. Liu, A Formal Software Design Language and Correctness Proof, *Proceeding of the Fifth Nordic Workshop on Programming Language (NORDIC'92)*, Tampere, Finland, Jan. 1992.

[80] S. D. Litvintchouk & A. S. Matsumoto, Design of Ada Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specification, in the book[18].

[81] M. Looijen and W. J. Kribbe, Management of Information Systems, Information System division, Faculty of Technical Informatics, Delft University of Technology, The Netherlands, 1989.

[82] M. Looijen, Management of Information Systems, *De Controller & Informatiemanagement*, Aug., 1992.

[83] M. D. Lubars, Environmental Support for Reuse, Technical Report, MCC, No. STP-120-88, 1988.

[84] M.D. McIlory, Mass-Produced Software components, *Software Engineering Concepts and Techniques*, 1968 NATO Conf. Software Eng., ed. J. M. Buxton, P. Naur, and B. Randell, pp. 88-98, 1976.

[85] B. Meyer, Genericity versus Inheritance, OOPSLA'86 Proceedings, September 1986, pp. 391-405.

[86] B. Meyer, Reusability: The Case for Object-Oriented Design. IEEE Software, V 4 (2), March 1987, pp. 50-64.

[87] G. Miller, The Magical Number Seven, Plus or minus two: Some Limits on our capability for Processing Information, *The Psychological Review*, Vol. 63(2), 1956, p.86.

[88] R. T. Mittermeir and M. Oppitz, Software Bases for the Flexible Composition of Application System, *IEEE Trans. on Software Engineering*, Vol. SE-13/4, Apr. 1987

[89] R. T. Mittermeir, Normalization of Software to enhance its Potential for Reuse, Institut fur Informatik, University of Klagenfurt, Austria.

[90] R. T. Mittermeir and E. Kofler, Layered Specifications to Support Reusability and Integratibility, *Proceeding of 2nd International Conference on System Integration*, IEEE Computer Science Press, June, 1992.

[91] J. Morris, A theoretical Basis for Stepwise Refinement and the programming Calculus, *Science of Computer programming 9*, North-Holland, 287-306, 1987.

[92] J. M. Neighbors, The Draco Approach to Constructing Software from Reusable Component, *IEEE Transaction on Software Engineering*, vol. 10, Sept., 1984.

[93] J. M. Neighbors, Draco: A Method for Engineering Reusable Software Systems, see also [18]. edited by T. J. Biggerstaff and A. J. Perlis, ACM press, Frontier Series, 1989.

[94] W. M. Osborne, E. J. Chikofsky, Fitting Pieces to the Maintenance Puzzle, *IEEE Software*, Jan., 1990.

[95] D. L. Parnas, P.C. Clements and D. M. Weiss, Enhancing Reusability with Information Hiding, see also [18].

[96] A.S. Peterson, Coming to Terms with Software Reuse Terminology: a Model-Based Approach *ACM SIGSOFT, Software Engineering Notes*, Vol 16, No. 2, Apr 1991, pp. 45-51.

[97] N. Plat, J. van Katwijk and K. Pronk, A case for Structured Analysis/Formal Design, Proceeding of VDM'91- Formal Software Development Methods, S. Prehn, W.J. Toetenel (editor), Springer-Verlag, LNCS 551, 1991.

[98] R. Prieto-Diaz, A software classification scheme. PhD thesis, Department of Information and computer science, University of California, Irvine, 1985.

[99] R. Prieto-Diaz and J. M. Neighbors, Module Interconnection Languages, *The Journal of Systems and Software*, 6, 307-337, 1986.

[100] R. Prieto-Diaz, P. Freeman, Classifying Software for Reusability. *IEEE Software*, V 4 (1), 6-16, Jan 1987.

[101] R. Prieto-Diaz, Making Software Reuse Work:An incremental Model, *ACM SIGSOFT Software Engineering Notes*, vol 16, No. 3, July, 1991.

[102] R. Prieto-Diaz, Implementing Faceted Classification for Software Reuse, *Communications of ACM*, May 1991, Vol.34, No.5, 89-97.

[103] N. J. Princeton, Cobol/MP Macro Facility Reference Manual version 9, Applied data research. 1979

[104] C. V. Ramamoorthy, V. Grarg, and A. Prakash, Support for Reusability in Genesis, *IEEE Tran. on Sof. Eng.*, Vol. 14, No. 8, 1988.

[105] G. P. Randell. Translating Data Flow Diagrams into Z (and Vice Versa). Technical Report 900019, Procurement Executive, Ministry of Defence, RSRE, Malvern, Worcestershire, UK, October 1990.

[106] S. Rugaber, S. B. Ornburn, and Jr. LeBlanc, Recognizing Design Decision in Programs, *IEEE Software,* January, 1990, pp. 46-54.

[107] M. Shaw, Abstraction Techniques in Modern Programming Languages, *IEEE Software*, Oct. 1984.

[108] M. Shaw, Prospects for an Engineering Discipline of Software, *IEEE Software*, November, 1990, pp. 15-24.

[109] B. Stroustrup, The C++ Programming Language-Reference Manual, AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

[110] H. Simon, Why should machines learn? in *Machine Learning*, Tioga, CA, 1983, pp. 25-38.

[111] B. Stroustrup, What is Object-Oriented Programming?, IEEE Software, May, 1988, pp. 10-20.

[112] H. G. Sol, *Simulation in Information Systems development*, Ph.D. Thesis, University of Groningen, The Netherlands, 1982.

[113] H. G. Sol, Dynamic Information Systems, *Dynamic Information Systems* H.G. Sol and R. L. Crosslin (Editors), Elsevier Science Publishers B. V. (North-Holland, 1991).

[114] W. F. Tichy, *Software Development Control Based on Systems Structure Description*, Ph. D. thesis, Carnegie-Mellon University, Computer Science Department, Jan. 1980.

[115] D. Thewlis, Programming Language, *Software Engineering Journal*, Vol. 1, No. 4, July 1986.

[116] W. Tracz, Software Reuse Myths, *ACM SISOFT, Software Engineering Notes*, Vol. 13, No. 1, Jan. 1988

[117] W. Tracz, Where does Reuse Start?, *ACM SIGSOFT, Software Engineering Notes*, Vol. 15, No. 2, 1990.

[118] W. Tracz, A conceptual Model for Megaprogramming, *Software Engineering Notes*, Vol. 16, No. 3, ACM SIGSOFT, July 1991. pp. 36-45.

[119] J. Tsai and J. Ridge, *Intelligent Support for Specifications Transformation.* IEEE Software, vol. 5(6), p. 34.

[120] F. Ververs, J. van Katwijk and L. Dusink, Direction in Reusing Software, Technical Report 88-58, Faculty of mathematics and ComputerScience, Delft University of Technology 1988.

[121] D. M. Volpanno, *STS-Software System, MCC Tech. Report* No. STP-257-87, July 29,87

[122] D. M. Volpano, The Template approach to Software Reuse, *MCC Tech. Report* No. STP-061-88, February, 11, 1988

[123] D. E. Webster, Design Representation Technology: A Survey, MCC Technical Report, No. STP-361-86, Oct-28-1986.

[124] D. E. Webster, Mapping the Design information representation Terrain, *Computer*, December, 1988.

[125] P. Wegner, Capital Intensive Software Technology, *IEEE Software*, July, 1984

[126] P. Wegner, Dimensions of Object-based Language Design, *OOPSLA'87 conference proceedings*, pp. 168-182, 1987

[127] G. M. Wijers, *Modelling Support in Information systems Development*, Ph.D. Thesis, Delft University of Technology, The Netherlands, 1991.

[128] N. Wirth, Algorithms + Data structures = programs Prentice-Hall, Englewood Cliffs, 1986.

Curriculum Vitae

Haikuan Li studied mathematics in the Mathematics Department of Beijing University in 1974 and graduated from this university in 1978. Then he worked and studied in the Computer Science Department, the Graduate School of Academia Sinica, Beijing, China. He completed the necessary courses in the field of computer science in the Graduate School and was a Teaching Assistant for several different courses in the field of computer science. In 1986 he was promoted as a lecturer of the Graduate School of Academia Sinica. He has several years of lecturing experience in teaching the graduate students of Academia Sinica.

Since 1988 Mr. Li, as a PhD candidate, has worked and studied in the Faculty of Mathematics and Computer Science, Delft University of Technology, the Netherlands. His research work was concerned with both software engineering and information systems. His research interest has been continually focused on reuse. The concepts, structures and models on reuse-in-the-large, as addressed by himself, regard to an information system, a support environment for application development and maintenance. This research resulted in twelve publications, including articles and technical reports.