Model Checking GOAL Agents

August, 2010

Sung-Shik T.Q. Jongmans

Model Checking GOAL Agents

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Sung-Shik T.Q. Jongmans
born in Seoul, Republic of Korea



Man-Machine Interaction Group
Department of Mediamatics
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

© 2010 Sung-Shik T.Q. Jongmans.

Model Checking GOAL Agents

Author: Sung-Shik T.Q. Jongmans
Student id: 1340263
Email: S.T.Q.Jongmans@student.tudelft.nl

Abstract

This thesis presents a novel approach to model checking of agent programs written in an agent programming language. The language we consider is GOAL. The novelty of the approach is that we implement model checking algorithms from scratch on top of the standard language interpreter. In contrast, in the literature on agent program verification, agents are translated to the input language of an existing model checker, after which this verification tool is used "as usual" for actual model checking.

We present an implementation, and show that our approach performs substantially better in terms of resource consumption than existing model checkers for GOAL. Moreover, our approach allows for more expressive property specification languages and has benefits from a software engineering point of view: the implementation of a language interpreter is gained "for free". Both conceptually and on the implementation level, our approach is generic and can be applied to other agent programming languages as well.

Additionally, we investigate state space reduction techniques tailored to GOAL, namely property-based slicing and partial order reduction, to further optimise our model checker's performance. To this end, we develop a static source code analysis method for GOAL agents. Case studies with both techniques show that substantial reductions can be gained when certain conditions are met.

Thesis Committee:

Chair:	Prof. Dr. C.M. Jonker, Faculty EEMCS, TU Delft
University supervisor:	Dr. K.V. Hindriks, Faculty EEMCS, TU Delft
	Dr. M.B. van Riemsdijk, Faculty EEMCS, TU Delft
Committee Member:	Dr. J.M. Broersen, Faculty of Science, Utrecht University

Preface

The 9th of November of 2009 was a memorable day in several ways. For one thing, it was my 22nd birthday. But also, it marked the start of my graduation project. Writing a preface seemed miles away at that time, and admittedly, it was. Getting from there to where I am now has been a long journey, sometimes fun, sometimes less, but always exciting, and I feel privileged to have been granted the opportunity to undertake it. As such, I would like to take a moment and thank the people who have helped me come this far.

First of all, I thank Koen and Birna for their supervision, support, feedback, critiques, comments, and reviews of my work. I am especially grateful for co-authoring [63] with me, and for teaching me how to conduct science and report on findings. I regard this as one of the most valuable lessons that I have learned over the past nine months.

Second, I thank Lăcrămioara Aştefănoaei and Louise Dennis for helping and providing me with, respectively, the MAUDE implementation of the GOAL interpreter and the AIL implementation of the GOAL interpreter, which are used in the performance comparison to be presented in Chap. 4. An additional "thank you" goes out to Lăcrămioara for reading and commenting on an early draft of parts of this thesis in preparation of [63].

Third, I thank Rob Hoogerwoord and Gerard Zwaan for introducing me to formal verification back in 2006 when I was still an undergraduate student in Eindhoven. Their lectures have definitely contributed to my enthusiasm for the subject.

I thank my parents for their love, support, education, and, of course, for owning a computer (though a Nintendo would have been nice too ;)).

Most of all, I thank Eva. You know, for everything.

Sung-Shik T.Q. Jongmans
Delft, the Netherlands
August 19, 2010

Contents

Preface iii

Contents v

I Core System 1

1 Prologue 3
1.1 Motivation . . . 3
1.2 Scope . . . 4
1.3 Contributions . . . 4
1.4 Organisation . . . 5

2 GOAL 7
2.1 Example . . . 7
2.2 Knowledge, Beliefs, Goals . . . 9
2.3 Mental State . . . 10
2.4 Actions . . . 11
2.5 Operational Semantics . . . 13
2.6 Summary . . . 14

3 Model Checker 17
3.1 Temporal Logic . . . 17
3.2 Model Checking . . . 20
3.3 Implementation . . . 24
3.4 Summary . . . 31

4 Comparison 33
4.1 Program Model Checking in Agent Verification . . . 33
4.2 Performance Comparison: Experimental Design . . . 36
4.3 Experiments and Results . . . 42
4.4 Discussion . . . 61
4.5 Summary . . . 63

II State Space Reduction 65

5 Transition Theory 67
5.1 Transitions . . . 67
5.2 Read / Write Sets . . . 69

5.3 Write Set Computation . . . 75
5.4 Relations on Transitions . . . 83
5.5 Summary . . . 92

6 Property-Based Slicing **93**
6.1 Slicing and Model Checking . . . 93
6.2 Influence . . . 95
6.3 Algorithm . . . 99
6.4 Case Studies . . . 101
6.5 Extensions . . . 108
6.6 Summary . . . 110

7 Partial Order Reduction **111**
7.1 Theory . . . 111
7.2 Implementation . . . 119
7.3 Case Studies . . . 123
7.4 Complementary Techniques . . . 130
7.5 Summary . . . 130

8 Epilogue **133**
8.1 Summary . . . 133
8.2 Conclusion . . . 134
8.3 Future Work . . . 134

III Appendices **137**

A Büchi Automata **139**
A.1 Program automata . . . 139
A.2 Property automata . . . 140
A.3 Product automata . . . 144
A.4 Language Emptiness . . . 147

B Experimental Results **149**

C Proofs **153**
C.1 Approximate Write Sets . . . 153
C.2 Independence Heuristics . . . 164
C.3 Slicing Algorithm . . . 167
C.4 Heuristics for **C1** . . . 174

Bibliography **179**

Part I

Core System

Chapter 1

Prologue

Over the years, software has become notorious for its proneness to programming errors. Although some of these errors are relatively harmless, others have caused serious damage. One infamous example in the area of space exploration is the first test flight of the Ariane 5 rocket in 1996. Due to an error in the rocket's control software, it veered off its flight path 37 seconds after launch, and was subsequently destroyed by its automated self-destruct system. The costs of this accident have been estimated at 370 million dollars [34].

Over the last decade, agent programming languages have started to find their way to safety-critical applications, e.g. autonomous spacecraft control [11]. Consequently, the need for techniques to *verify* such agent systems, tailored to the agent-oriented programming paradigm [87], has risen. *Verification* is the process of establishing that a system is correct with respect to its stated requirements. The current agent literature primarily focusses on verification by *model checking*, a method proposed in the early 1980s as an alternative to verification by mathematical proofs. In this thesis, we continue the work in this area by developing a model checker for the Goal agent language, introduced in [53].

Our approach, however, deviates from existing efforts quite substantially. In past research on agent model checking, *existing* model checkers are used for actual verification: the agent program written in an agent language (e.g. AgentSpeak) is encoded in the input language of an existing model checker, after which the model checker is "run as usual". This has the benefit that model checking algorithms that already have been implemented in such tools can be reused at the cost of reimplementing the language's semantics in the model checker's input language. In this thesis, we take the opposite approach: we implement model checking algorithms from scratch, but reuse the existing implementation of the semantics, i.e. the standard interpreter.

1.1 Motivation

Reusing existing tools in a new problem domain is often tempting: the existing technology has already proven itself, and developing a new tool from scratch may be daunting or irrational. For example, the Ada programming language, although originally developed for use in military projects by the US Department of Defence, has found a niche outside the military in other safety-critical applications, e.g. commercial aviation projects.

Thus, the idea of using existing technology for model checking agent programs is not a bad idea at all. We believe, however, that choosing an existing model checker as the tool to be reused (rather than an existing interpreter) may not be the best. The literature has already identified one issue with this approach: encoding the agent language's semantics in the input language of an existing model checker is complex. This is a problem from a software engineering point of view. In addition, we speculate that there are also performance issues: although existing tools have been optimised for model checking of imperative languages, these optimisations may not work well on translated agent programs. Moreover, by translating a high-level agent program to imperative source code that an existing model checker can handle, verification occurs on a lower

level of abstraction than necessary.

By reusing existing model checkers rather than existing interpreters, one tacitly assumes that developing a model checker is more "difficult" (in terms of implementation and optimisation) than developing an interpreter. We believe the converse. Testing this hypothesis has been the primary motivation for the first half of this thesis.

The second half of the thesis focusses on the application of optimisation techniques that are well-known from "traditional" model checking to model checking of agent programs. Up to now, the adaptation of such methods to agents has been limited to only three publications, while there is no reason to believe that optimisation in agent model checking is less necessary than for traditional model checking.

1.2 Scope

The thesis focusses on the verification of *concrete agent programs* written in an actual agent programming language. We do not consider verification of *abstract agent models*, which we see as a different line of research within the agent model checking field (e.g. [83]).

The agent programming language that we consider is Goal, introduced in [53]. Goal is a language actively being developed at TU Delft. Although choosing Goal as the agent language under investigation as such offered geographical benefits (the thesis was written in Delft), another important consideration has been that two model checkers for Goal, based on existing tools, already existed before our work commenced, enabling a quantitative performance comparison with our model checker to be presented.

Contrary to the existing literature and projects, this thesis focusses on single-agent systems in the absence of an environment. It is our philosophy that we first must find a means to verify such systems *efficiently* before focussing attention on the more complex multi-agent system case. As our experiments will demonstrate, current state-of-the-art is not yet able to verify quite simple single-agent systems, supporting our choice in this respect.

1.3 Contributions

We identify the following main contributions of this thesis.

- *Interpreter-based model checker for* Goal

  We implemented model checking algorithms from scratch on top of the standard Goal interpreter. As outlined, such an approach to agent model checking has not been taken before.

- *Model checking framework in* Java

  Because we wanted our model checker to be highly flexible, the implementation in Java has a high degree of modularity. As such, an interpreter-based model checker for another agent language with an interpreter written in Java (e.g. AgentSpeak with Jason) can easily be developed with this framework.

- *Binary mental states*

  We have developed an efficient bit string representation of *mental states*, the primary data structure of Goal agents, that allow for fast comparison (a frequent operation in model checking) and require little space to be stored. Such representations of states are not used in other agent model checkers.

- *Quantitative performance comparison*

  To test our hypothesis that our interpreter-based approach indeed works more efficiently than approaches in which an existing model checker is used, we have compared our model checker with existing Goal model checkers in an extensive survey. Such an in-depth comparison has not been carried out before.

- *State space reduction techniques*

  We have implemented two optimisations known from imperative language model checking, called *state space reduction techniques*: *property-based slicing* and *partial order reduction*. The former method has earlier been applied in agent verification to AgentSpeak programs, but differs considerably from our approach in the algorithm and concepts on which it is based. The latter method has not been used for model checking agent programs yet.

- *Static source code analysis method*

  To implement aforementioned state space reduction techniques, we needed additional theory about the execution of a Goal agent. Specifically, we needed a means to establish how an action that an agent performs in the present can influence its ability to perform actions in the future by mere static analysis of the source code. The theory of how this should be done was not yet available for Goal, nor have we found similar theory in the agent literature.

1.4 Organisation

The thesis is organised as follows. In the first part, covering Chaps. 1,2,3,4, we discuss the core model checker. First, in Chap. 2, we introduce the Goal agent language. The example that we use to illustrate Goal's concepts will reoccur in subsequent chapter as well. In Chap. 3, we first treat a logical language in which desired properties about Goal agent can be expressed. Subsequently, we introduce model checking as a decision procedure for determining whether an agent satisfies such a property. Finally, we discuss how we implemented the theory for Goal, and describe the architecture of the system. Chapter 4 compares our approach to Goal model checking with the literature, and presents a performance evaluation along with some issues we encountered during experimental design. An abridged version of these first chapters has been published as [63].

In the second part, covering Chaps. 5,6,7, we discuss state space reduction techniques that extend the core model checker (note that these optimisations have not been used in our performance comparison of Chap. 4). In Chap. 5, we first introduce additional theory about the execution of a Goal agent based on static analysis of its source code. Next, in Chap. 6, we treat property-based slicing; in Chap. 7, we treat partial order reduction. Both techniques are exemplified with case studies that shed light on their ability to reduce resource consumption. The thesis is concluded in Chap. 8. The appendices consist of additional explanation about the model checking theory that we apply, experimental results, and supporting propositions and lemmas on which theorems in the main text are built.

Related work is discussed during the exposition of the subject to which it is related, instead of recorded separately in a distinct chapter or section.

Chapter 2

GOAL

This chapter treats the GOAL programming language. GOAL, introduced nearly a decade ago in [53], is an acronym for *goal-oriented agent language*. Indeed, GOAL is a language in which rational agents can be defined, and facilitates programming according to the agent-oriented programming paradigm [87]. An important difference between GOAL and other such languages, e.g. AGENTSPEAK, is that GOAL agents have *goals-to-be* (i.e. *declarative goals*) rather than *goals-to-do*.

The treatise in this chapter is derived from [53, 30, 51, 52], and discusses the non-environment single-agent subset of GOAL (in compliance with the scope of this thesis). We proceed as follows. In Sect. 2.1, we introduce the GOAL language with an example agent. Then, in Sect. 2.2, we discuss how knowledge, beliefs and goals are represented. Section 2.3 treats the concept of a mental state, and in Sect. 2.4, actions are discussed. Section 2.5 presents the operational semantics of a GOAL agent. Finally, Sect. 2.6 summarises the chapter.

2.1 Example

We introduce the GOAL language with an example: meet `blenderAgent`, whose source code is given in Fig. 2.1. `blenderAgent`'s task is to put fruit, namely bananas and oranges, into a blender to make juice. It works as follows. First, `blenderAgent` needs to wash the fruit it is about to put into the blender's reservoir. Subsequently, it adds a number of bananas and oranges to this reservoir according to some recipe. If sufficiently many bananas or oranges have been added, this type of fruit is ticked off by means of a check box on the recipe. Finally, if all fruit is added, i.e. all check boxes are ticked, `blenderAgent` starts the blender.

The GOAL code by which `blenderAgent` is defined is divided into 5 different sections; we discuss each of them briefly below, and treat the details in subsequent sections.

Knowledge Section (lines 2-7) The **knowledge** section of a GOAL agent contains all the information that it has about the domain it is acting in, and of which it *knows* that it always will be true. That is, knowledge is *static*, meaning that it does not change during execution. For example, one of the pieces of knowledge that `blenderAgent` has is that the recipe it is following prescribes the inclusion of 2 bananas (line 3). As this is in `blenderAgent`'s **knowledge** section, the recipe remains unchanged during execution. The knowledge in the **knowledge** section of a GOAL agent is commonly referred to as its *knowledge base*.

Beliefs Section (lines 8-12) The beliefs of a GOAL agent are, in contrast to its knowledge, *dynamic*, meaning that they can change during execution. For example, initially, `blenderAgent` believes that it has added 0 bananas to the blender's reservoir (line 9). Of course, if `blenderAgent` is free of bugs, we expect it to at some point believe that it has added 2 bananas to the reservoir in accordance with the recipe. The collection of beliefs of a GOAL agent is commonly referred to as its *belief base*.

Goals Section (lines 13-15) The goals of a GOAL agent represent a state of the world that it wants to bring about. For example, `blenderAgent` wants to achieve that the blender's

```
1  main: blenderAgent{
2    knowledge{
3      recipe(bananas,2).
4      recipe(oranges,2).
5      toAdd(F) :- recipe(F,Qr), added(F,Q), Q<Qr.
6      filled :- ticked(bananas), ticked(oranges).
7    }
8    beliefs{
9      added(bananas,0).
10     added(oranges,0).
11     switch(off).
12   }
13   goals{
14     filled, switch(on).
15   }
16   program{
17     if goal(filled), bel(not(washed)) then wash.
18     if goal(filled), bel(toAdd(bananas)) then add(bananas,1).
19     if goal(filled), bel(toAdd(oranges)) then add(oranges,1).
20     if goal(filled), bel(recipe(F,_)) then tick(F).
21     if goal(switch(on)), bel(filled) then blend.
22   }
23   actionspec{
24     add(F,Qinc) {
25       pre{ washed, added(F,Q), Qnew is Q+Qinc }
26       post{ not(added(F,Q)), added(F,Qnew) }
27     }
28     blend{
29       pre{ switch(off) }
30       post{ not(switch(off)), switch(on) }
31     }
32     tick(F){
33       pre{ not(ticked(F)), washed, recipe(F,Qr), added(F,Qr) }
34       post{ ticked(F) }
35     }
36     wash{
37       pre{ not(washed) }
38       post{ washed }
39     }
40   }
41 }
```

Figure 2.1: Example GOAL agent, called `blenderAgent`, whose task is to put fruit into a blender.

reservoir is filled, and that the blender is switched on. Note the declarative nature of how goals are represented: `blenderAgent`'s goal is not specified as the performance of a sequence of actions that results in the blender being filled, but as a description of what the state of the world should be like. The collection of goals of a GOAL agent is commonly referred to as its *goal base*.

Program Section (lines 16-22) The **program** section of a GOAL agent contains *action rules* that the agent uses to decide on which action to perform. An action rule specifies that if some expression about the current beliefs and goals of the agent is true, then some action may be considered for performance. For example, the action rule on line 17 of `blenderAgent`'s code specifies that if `blenderAgent` has the goal that the blender is filled and believes that it has not yet washed the fruit, then it may do so. We refer to the collection of action rules in the **program** section of a GOAL agent as its *rule base*.

Actionspec Section (lines 23-40) Finally, a GOAL agent has a series of action specifications that specify when an action can be performed, and what the results of performing the action are. The former is called an action's *precondition*, and the latter is called its *postcondition*. The difference between an action rule and an action's precondition is that the former specifies when it is a *good idea* to perform the action, whereas the latter specifies when an action *can* be performed. The actions that are specified in the **actionspec** section of the agent are called *user-defined actions*. In addition, GOAL also has a number of *built-in actions* (see Sect. 2.4).

2.2 Knowledge, Beliefs, Goals

To express the knowledge, beliefs, and goals of a GOAL agent, we need a formalism in which they can be specified. The current implementation of GOAL uses *Prolog* (PL) for this purpose. We assume familiarity with Prolog (an introduction is given in [16]).

The knowledge base of an agent is a set of PL clauses, i.e. rules and facts, denoted by K. Every clause can be thought of as a piece of knowledge that the agent has about the world it is acting in. The knowledge base of `blenderAgent` contains two facts, namely `recipe(bananas,2)` and `recipe(oranges,2)`, and two rules:

```
toAdd(F) :- recipe(F,Qr), added(F,Q), Q<Qr.
filled :- ticked(bananas), ticked(oranges).
```

The two facts constitute the `recipe/2` predicate, which expresses how many pieces of each type of fruit should be added to the blender. The former rule constitutes the `toAdd/1` predicate, and expresses that some piece of fruit `F` needs to be added to the reservoir (namely if the number of `F`s that have already been added to it is less than what the recipe prescribes). The latter rule constitutes the `filled/0` predicate, and expresses that the blender is sufficiently filled if all types of fruit on the recipe have been ticked off.

The belief base of an agent is, similar to its knowledge base, a set of PL clauses, denoted by Σ. There are, however, two differences. The first is that the knowledge base may contain any PL clause, whereas the belief base is restricted to ground facts only.[1] The second difference is that, once initialised, clauses may not be added to or removed from the knowledge base, whereas the contents of the belief base is subject to constant change. The initial belief base of `blenderAgent` contains three ground facts: `added(bananas,0)`, `added(oranges,0)`, and `switch(off)`. The `added/2` predicate is used to keep track of which pieces of fruit have already been added to the reservoir; the `switch/1` predicate indicates that the blender is switched on or off.

Finally, the goal base of an agent, denoted by Γ, is a set of conjunctions of ground facts. Each such conjunction represents a single goal γ that the agent has at some instant. The goals in the goal base may change over time, whereas a single goal cannot be changed once adopted. `blenderAgent`'s **goals** section defines only one single goal:

```
filled, switch(on).
```

That is, `blenderAgent`'s initial goal base is $\Gamma = \{\gamma\} = \{$`filled, switch(on)`$\}$ (note that "`,`" denotes PL's conjunctive operator, while "," is used as separator for elements in a set). Alternatively, one could have made `filled` and `switch(on)` two individual goals. In that case, the **goals** section of `blenderAgent`'s code would have looked as follows:

```
filled.
switch(on).
```

Note that the "`,`" behind `filled` is substituted for a "`.`" (the line break has no meaning).

Are there semantic differences between these two goal bases, and if so, which one should we choose? With respect to the first question, yes, semantically there are differences. To the second question, however, there is no universal answer: it depends on *when* the agent should bring about its goals. Informally, all the conjuncts of a single goal represent a (sub-)state of the world that the agent wants to bring about *simultaneously*. Thus, goal γ expresses that `filled` and `switch(on)` need be true at the same time for the goal to be achieved. This is, in our view, the correct behaviour of `blenderAgent`. In contrast, the combination of γ_1 and γ_2 expresses that `blenderAgent` wants to bring about `filled` and `switch(on)` at some future point in time, but not necessarily simultaneously. This means that it might first switch on the blender, and thereafter put the pieces of fruit into its reservoir. This is not the behaviour that we would expect of `blenderAgent`.

[1] In principle, GOAL also allows the inclusion of PL rules in the belief base. For simplicity of later expositions and without loss of generality, however, we assume that all such rules are in the knowledge base.

2.3 Mental State

Let us have a more formal look at how the syntactic differences between knowledge, beliefs, and goals are reflected in their semantics. First, we introduce the concept of a *mental state*. The mental state, denoted by μ, is the primary data object of a GOAL program. It is comprised of the knowledge, beliefs, and goals of the agent at some point during its execution, and represented as the tuple $\mu = \langle K, \Sigma, \Gamma \rangle$. The *initial mental state* of a GOAL agent, typically denoted by μ_0, is the mental state with which execution of the agent commences, and specified by the **knowledge**, **beliefs**, and **goals** sections of its code. For example, the initial mental state of `blenderAgent` looks as follows:

$$
\begin{array}{lcll}
K & = & \{ & \texttt{recipe(bananas,2)} \quad , \\
 & & & \texttt{recipe(oranges,2)} \quad , \\
 & & & \texttt{toAdd(F) :- recipe(F,Qr), added(F,Q), Q<Qr} \quad , \\
 & & & \texttt{filled :- ticked(bananas), ticked(oranges)} \quad \} \\
\Sigma & = & \{ & \texttt{added(bananas,0)}, \texttt{added(oranges,0)}, \texttt{switch(off)} \quad \} \\
\Gamma & = & \{ & \texttt{filled, switch(on)} \quad \}
\end{array}
$$

2.3.1 Mental State Conditions

A GOAL agent decides on the actions it performs based on the contents of its mental state. Consequently, it needs the ability to inspect its own beliefs and goals. This ability is captured by *mental state conditions* (MSC). An MSC is an expression that comes from the *language of mental state conditions*, denoted by $\mathcal{L}_\Psi$, and whose evaluation results in true or false. Examples of MSCs that appear in the code of `blenderAgent` are the following (in order of occurrence): **goal**`(filled)`, **bel**`(not(washed))`, **bel**`(toAdd(bananas))`, **bel**`(toAdd(oranges))`, **bel**`(recipe(F,_))`, **goal**`(switch(on))`, and **bel**`(filled)`. In general, the syntax of an MSC, denoted by ψ, is defined as follows.

$$
\begin{array}{lcl}
\chi & ::= & \text{any PL query} \\
\psi & ::= & \mathbf{bel}(\chi) \mid \mathbf{goal}(\chi) \mid \neg\psi \mid \psi \wedge \psi
\end{array}
$$

We call **bel** the *belief operator*, and **goal** the *goal operator* ($\neg$ and $\wedge$ are operators for negation and conjunction as usual).

The truth or falsehood of MSCs is defined relative to the current mental state μ by means of the entailment relation $\models_\Psi$. Informally, $\mathbf{bel}(\chi)$ is true if χ is believed by the agent. To establish whether χ is a belief or not, the PL query χ is evaluated in a PL *database* that contains all the clauses that are in the current belief base (only ground facts) and the static knowledge base (both facts and rules). If such a query results in a list of substitutions, this is interpreted as "true", i.e. χ is indeed a belief of the agent. Otherwise, χ is assumed not to be believed. Determining the truth or falsehood of a $\mathbf{goal}(\chi)$ MSC is slightly more involved, because the goal base is a set of conjunctions of facts, rather than a set of facts like the belief base. Informally, $\mathbf{goal}(\chi)$ is true if there exists a goal γ in the goal base of which χ is a (sub-)goal. Determining whether χ is a (sub-)goal of γ is done similar to the evaluation of a **bel** formula: χ is evaluated in a PL database containing all the conjuncts (only ground facts) that occur in γ, combined with the knowledge base.

We denote the evaluation of a PL query in a PL database symbolically by the entailment relation $\models$, i.e. if X is a set of clauses and χ is a query, then $X \models \chi$ denotes the evaluation of χ in the PL database containing all elements of χ. For convenience but with a slight abuse of notation, we will write $\gamma \models \chi$ if γ is a conjunction of facts $\gamma = \chi_0\texttt{,}\ldots\texttt{,}\chi_n$ in which case we actually should write $\{\chi_0, \ldots, \chi_n\} \models \chi$ (note the difference between the use of PL's conjunctive operator "`,`", and the separator for elements in a set ","). We summarise the above exposition formally as follows.

Definition 1. *Let $\mu = \langle K, \Sigma, \Gamma \rangle$ be a mental state.*
Then:

$$
\begin{array}{lll}
\mu \models_\Psi \textbf{bel}(\chi) & \text{iff} & K \cup \Sigma \models \chi \\
\mu \models_\Psi \textbf{goal}(\chi) & \text{iff} & \text{there exists a } \gamma \in \Gamma \text{ such that } K \cup \gamma \models \chi \\
\mu \models_\Psi \neg\psi & \text{iff} & \mu \not\models_\Psi \psi \\
\mu \models_\Psi \psi_1 \wedge \psi_2 & \text{iff} & \mu \models_\Psi \psi_1 \text{ and } \mu \models_\Psi \psi_2
\end{array}
$$

All the MSCs that occur in `blenderAgent` (listed above) are true in its initial mental state. For example, **goal(**`filled`**)** is true, because the evaluation of the query `filled` in the PL database corresponding to the union of the (only) goal $\gamma =$ `filled, switch(on)` and the knowledge base K results in a singleton list containing the empty substitution, i.e. `[[]]`. As another example, consider the MSC **bel(**`recipe(F,_)`**)**, which requires the evaluation of the PL query `recipe(F,_)` in $\Sigma \cup K$. Due to the definition of the `recipe/2` predicate in the knowledge base, this query results in the list of substitutions `[[F/bananas],[F/oranges]]` such that **bel(**`recipe(F,_)`**)** is true.

2.3.2 Terminology and Notation

In the remainder of this and subsequent chapters, we adopt the following terminology and notation. A *literal* is a PL fact or its negation. In the former case, the literal is called *positive*, and in the latter case *negative*. If a PL term χ is a conjunction of literals, then $literals^+(\chi)$ denotes the set of positive literals, and $literals^-(\chi)$ denotes the set of negative literals.

The set of free variables that occur in a PL term χ (e.g. a query in an MSC, or the body of a rule in the knowledge base) is denoted by $free(\chi)$. For example, let $\chi =$ `added(F,Q)`; then $free(\chi) = \{$`F`, `Q`$\}$. A PL substitution is denoted by θ, and the set of variables that it binds is denoted by $domain(\theta)$. For example, let $\theta =$ `[F/bananas]`; then $domain(\theta) = \{$`F`$\}$. Application of a substitution to a term χ is denoted by $\chi\theta$, e.g. $\chi\theta =$ `added(bananas,Q)` (note that $free(\chi\theta) = \{$`Q`$\}$), and composition of substitutions θ and θ' is denoted by $\theta \circ \theta'$. For example, let $\theta' =$ `[Q/1]`; then $\theta \circ \theta' =$ `[F/bananas,Q/1]`. The empty substitution is denoted by `[]`. The set of all PL terms is denoted by $\mathcal{L}$, and the subset of $\mathcal{L}$ that can be derived from the contents of some PL database X is called X's *theory*, and denoted by $theory(X)$, i.e. $theory(X) = \{\chi \in \mathcal{L} \mid X \models \chi\}$.

Additionally, a *mental atom* is a **bel**(χ) formula or a **goal**(χ) formula. We denote the set of all mental atoms in an MSC ψ by $Atoms(\psi)$. A *mental literal* is a mental atom (positive) or its negation (negative). Because mental atoms may contain PL queries, they may contain PL variables. We denote all variables that occur in an MSC ψ by $free(\psi)$. The application of a substitution θ to an MSC ψ yields the MSC that results from applying θ to all the PL queries that occur in ψ. For example, let $\psi =$ **bel(**`toAdd(F)`**)** and $\theta =$ `[F/bananas]`. Then, $\psi\theta =$ **bel(**`toAdd(bananas)`**)**.

2.4 Actions

To achieve its goals, an agent needs to select and perform actions. Because environments are not under consideration in this thesis, actions only affect the agent's mental state. As mentioned, we distinguish two classes of actions: user-defined actions and built-in actions. We denote an action by α.

In the GOAL code of an agent, user-defined actions are defined in the **actionspec** section. Each such action has an identifier with a number of parameters (e.g. `add(F,Qinc)`), a precondition and a postcondition. Mathematically, we represent this as the tuple $\alpha = \langle \chi_{\texttt{pre}}, \chi_{\texttt{post}} \rangle$, where $\chi_{\texttt{pre}}$ is the precondition, and $\chi_{\texttt{post}}$ the postcondition. More specifically, $\chi_{\texttt{pre}}$ is a PL query that specifies the beliefs that the agent must have for the action to be performable. Thus, we can regard the precondition of a user-defined action as the MSC **bel**$(\chi_{\texttt{pre}})$. The postcondition is a conjunction of literals, and represents the transformations to the belief base that are brought about if the action is performed: every positive literal is added to the belief base, whereas every negative literal is removed from it. We denote this operation by $\oplus$, i.e. if Σ is the belief base, and χ is a conjunction of literals, then the updated belief base Σ' is defined as follows:

$$
\begin{array}{rcl}
\Sigma' & = & \Sigma \oplus \chi \\
& = & (\Sigma \cup literals^+(\chi)) \setminus literals^-(\chi)
\end{array}
$$

In addition to these updates to the belief base, all the goals that are completely achieved after updating the belief base are removed from the goal base.

GOAL's built-in actions are **adopt**(χ), **drop**(χ), **insert**(χ), **delete**(χ), and **skip**. An **adopt**(χ) action, in which χ is a conjunction of ground facts, adds χ as a new goal to the goal base. Implicit precondition of an **adopt**(χ) action is that χ is not already believed by the agent (it is considered irrational for an agent to adopt a goal that has already been brought about). GOAL does not require a goal to be adopted to be consistent with the goals already in the goal base. For example, in addition to the goal `filled, switch(on)`, `blenderAgent` may very well adopt the goal `switch(off)`. Although it is impossible for the blender to be switched on and off simultaneously, `blenderAgent` can achieve both goals one after the other. That is, they should be read as "eventually `filled, switch(on)`" and "eventually `switch(off)`", which resolves the inconsistency. This temporal dimension is not made explicit in the notation of goals, but embedded in GOAL's semantics. A **drop**(χ) action, in which χ is a conjunction of positive literals, removes all goals from the goal base of which χ can be derived. The other three actions are abbreviations for $\langle \texttt{true}, \chi \rangle$, $\langle \texttt{true}, \texttt{not}(\chi) \rangle$, and $\langle \texttt{true}, \texttt{true} \rangle$, respectively, in which χ is a conjunction of positive literals. Therefore, we will not discuss **insert**, **delete**, and **skip** explicitly in the remainder of the thesis.

When an action is performed, the mental state is *transformed*. To formalise this change, GOAL implements a function, called the *mental state transformer* (MST), that maps an action α and a mental state μ to μ's *successor*, i.e. the state that results from executing α in μ. The MST, denoted by $\mathcal{M}$, is defined as follows.

Definition 2. *Let α be an action, and let $\mu = \langle K, \Sigma, \Gamma \rangle$ be a mental state. Then:*

$$\mathcal{M}(\alpha, \mu) = \begin{cases} \langle K, \Sigma', \Gamma' \rangle & \text{if } \alpha = \langle \chi_{\texttt{pre}}, \chi_{\texttt{post}} \rangle \\ & \text{and there exists a substitution } \theta \text{ such that } \mu \models_\Psi \textbf{bel}(\chi_{\texttt{pre}}\theta) \\ & \text{and } free(\chi_{\texttt{post}}\theta) = \emptyset \\ & \text{and } \Sigma' = \Sigma \oplus \chi_{\texttt{post}}\theta \\ & \text{and } \Gamma' = \Gamma \setminus theory(K \cup \Sigma') \\ \langle K, \Sigma, \Gamma' \rangle & \text{if } \alpha = \textbf{adopt}(\chi) \\ & \text{and } \mu \not\models_\Psi \textbf{bel}(\chi) \\ & \text{and } free(\chi) = \emptyset \\ & \text{and } \Gamma' = \Gamma \cup \{\chi\} \\ \langle K, \Sigma, \Gamma' \rangle & \text{if } \alpha = \textbf{drop}(\chi) \\ & \text{and } \Gamma' = \Gamma \setminus \{\gamma \in \Gamma \mid K \cup \gamma \models \chi\} \end{cases}$$

We have already discussed most of the definition of the MST informally above. Exceptions are the conditions $free(\chi_{\texttt{post}}\theta) = \emptyset$ (if α is a user-defined action) and $free(\chi) = \emptyset$ (if α is an **adopt** action). The idea behind these conditions is that updates to Σ and Γ may not contain facts in which free variables occur, because the belief base and goals in the goal base may only contain ground facts.

To bind the free variables that might occur in $\chi_{\texttt{post}}\theta$ and γ, substitutions can be applied to actions as follows.

$$\alpha\theta = \begin{cases} \langle \chi_{\texttt{pre}}\theta, \chi_{\texttt{post}}\theta \rangle & \text{if } \alpha = \langle \chi_{\texttt{pre}}, \chi_{\texttt{post}} \rangle \\ \textbf{adopt}(\chi\theta) & \text{if } \alpha = \textbf{adopt}(\chi) \\ \textbf{drop}(\chi\theta) & \text{if } \alpha = \textbf{drop}(\chi) \end{cases}$$

The application of the MST to an action α and mental state μ may then look as $\mathcal{M}(\alpha\theta, \mu)$. This formalises how arguments are bound to the parameters that occur in the identifier of an action specification in the GOAL code. For example, to *instantiate* the $\texttt{tick(F)} = \langle \chi_{\texttt{pre}}, \chi_{\texttt{post}} \rangle$ action such that `F` is bound to `bananas`, we write the following. Let $\chi_{\texttt{post}} = \texttt{ticked(F)}$.

$$\begin{aligned}\langle \chi_{\texttt{pre}}, \chi_{\texttt{post}} \rangle \texttt{[F/bananas]} &\equiv \langle \chi_{\texttt{pre}}, \texttt{ticked(F)} \rangle \texttt{[F/bananas]} \\ &\equiv \langle \chi_{\texttt{pre}} \texttt{[F/bananas]}, \texttt{ticked(F)[F/bananas]} \rangle \\ &\equiv \langle \chi_{\texttt{pre}} \texttt{[F/bananas]}, \texttt{ticked(bananas)} \rangle\end{aligned}$$

Note that the mental state transformer is a *partial function*, i.e. it is not defined for all pairs of actions and mental states. If $\mathcal{M}$ is defined for an action α and a mental state μ, then we call α *enabled* in μ; only actions that are enabled in μ are performable in μ. Although enabledness of actions expresses when an action *can* be performed, it says nothing about when it is a *good idea* to perform an action. Indeed, if a GOAL agent would always choose and perform an action from the set of all its enabled actions at random, this would only rarely (if ever) lead to rational behaviour.

Action rules work as a constraint on the enabled actions that the agent may perform: they define the *strategy* or *policy* that the agent applies to achieve its goals. Mathematically, the **program** section of an agent is represented as a set R and each $\rho \in R$ is of the form $\rho =$ **if** ψ **then** α. An action rule is called *applicable* if its MSC ψ is true. If an action α is both enabled and there exists an applicable action rule of which α is the consequent, then α is called an *option*. Action rules are said to *generate* options, and an agent only performs actions that are also options.

2.5 Operational Semantics

Informally, a GOAL agent executes as follows. First, it establishes which rules in its rule base are applicable by evaluation of their MSC in the current mental state. Then, it generates its options based on these applicable rules. Finally, it non-deterministically chooses one of these options to actually perform. The process repeats itself until no more rules are applicable, after which the agent performs the **skip** action indefinitely. Thus, the execution of a GOAL agent is always infinite.

We express the above by the following transition rule, which describes how an agent gets from one mental state to another.

Definition 3. *Let μ be a mental state, let* **if** ψ **then** α *be an action rule, and let θ be a substitution. The transition relation $\longrightarrow$ is the smallest relation induced by the following transition rule:*

$$\frac{\mu \models_\Psi \psi\theta \quad \mathcal{M}(\alpha\theta, \mu) \text{ is defined}}{\mu \longrightarrow \mathcal{M}(\alpha\theta, \mu)}$$

Note that the evaluation of the MSC ψ in an action rule **if** ψ **then** α is used to instantiate α. For example, recall that evaluation of the the mental literal **bel**(`recipe(F,_)`) in the MSC ψ of the action rule **if goal**(`filled`), **bel**(`recipe(F,_)`) **then** `tick(F)` in the initial mental state μ_0 of `blenderAgent` results in a list of two substitutions, namely `[[F/bananas],[F/oranges]]`. Thus, there exist two θs such that $\mu_0 \models_\Psi \psi\theta$, namely $\theta =$ `[F/bananas]` and $\theta =$ `[F/oranges]`. Consequently, the `tick(F)` action can be instantiated with `bananas` or `oranges`. (As a side note, we remark that neither of these instantiations of `tick(F)` are also options in μ_0, because their precondition is not satisfied: `washed` is not yet a belief of `blenderAgent` in μ_0.)

Given the transition relation $\longrightarrow$, we define the semantics of a GOAL agent by means of a *transition system*. Let the transition system $\mathcal{T}$ of a GOAL agent be the tuple $\langle \Omega_M, \mu_0, \longrightarrow \rangle$, in which Ω_M is the set of all mental states that the agent can be in, μ_0 is the initial mental state, and $\longrightarrow$ is the transition relation that connects mental states. We call Ω_M the *state space* of the agent, whose *size* is the number of mental states in Ω_M. We assume that the state space of an agent is *finite*, i.e. its size is bounded by some finite natural number. This assumption is important because model checking algorithms only terminate if the state space of the program under investigation is finite.[2]

We can regard $\mathcal{T}$ as a digraph in which Ω_M is the set of vertices, μ_0 is some distinguished vertex, and $\longrightarrow$ is the set of edges. Then, a *path* π in $\mathcal{T}$ is a, possibly infinite, sequence of mental states $\pi = \mu\mu'\mu'' \cdots$ such that for each two consecutive mental states $\mu^{(i)}\mu^{(i+1)}$ holds

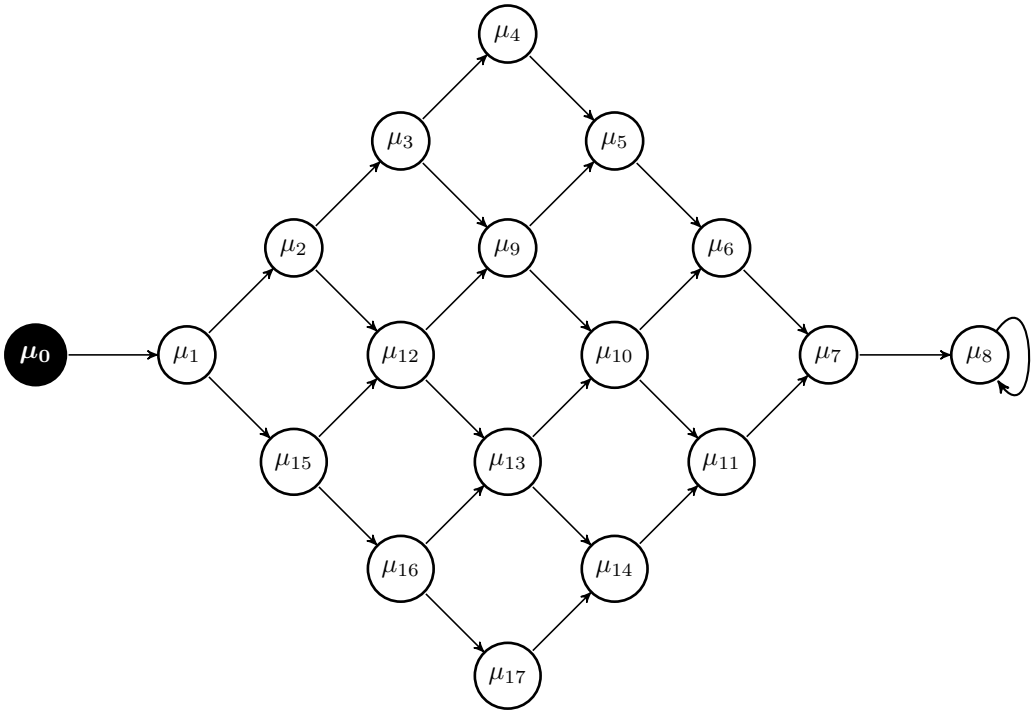
[2]The assumption is a restriction that we impose on the GOAL agents to which this thesis applies, and is not inherent to GOAL. That is, in general, the state space of a GOAL agent may be infinite.

$\mu^{(i)} \longrightarrow \mu^{(i+1)}$. Let π_i denote the i-th mental state on path π. A *computation*, denoted $\boldsymbol{\pi}$, is a distinguished infinite path in $\mathcal{T}$ such that $\boldsymbol{\pi}_0 = \mu_0$. The semantics of a GOAL agent is defined as the set of computations, i.e. all paths in its transition system starting from the initial mental state.

The transition system of `blenderAgent` appears, as a digraph, in Fig. 2.2. `blenderAgent`'s state space is of size 18, and there are 16 distinct paths through the system that start in μ_0, i.e. `blenderAgent` has 16 different computations.

2.6 Summary

In this chapter, we have treated the GOAL agent language that is under consideration in the remainder of the thesis. We started with an exposition on how knowledge, beliefs, and goals are represented, continued with the notion of mental states, proceeded with a treatise of actions, and ended with GOAL's semantics. The running example with which we illustrated GOAL's constructs involved `blenderAgent`; this agent will reappear quite often in future chapters. The theory presented here is already well-established GOAL material, and, as such, this chapter did not form a contribution to the existing literature.



μ_0 {added(bananas,0), added(oranges,0), switch(off)}
μ_1 {added(bananas,0), added(oranges,0), switch(off), washed}
μ_2 {added(bananas,1), added(oranges,0), switch(off), washed}
μ_3 {added(bananas,2), added(oranges,0), switch(off), washed}
μ_4 {added(bananas,2), added(oranges,0), switch(off), washed, ticked(bananas)}
μ_5 {added(bananas,2), added(oranges,1), switch(off), washed, ticked(bananas)}
μ_6 {added(bananas,2), added(oranges,2), switch(off), washed, ticked(bananas)}
μ_7 {added(bananas,2), added(oranges,2), switch(off), washed, ticked(bananas), ticked(oranges)}
μ_8 {added(bananas,2), added(oranges,2), washed, ticked(bananas), ticked(oranges), switch(on)}
μ_9 {added(bananas,2), added(oranges,1), switch(off), washed}
μ_{10} {added(bananas,2), added(oranges,2), switch(off), washed}
μ_{11} {added(bananas,2), added(oranges,2), switch(off), washed, ticked(oranges)}
μ_{12} {added(bananas,1), added(oranges,1), switch(off), washed}
μ_{13} {added(bananas,1), added(oranges,2), switch(off), washed}
μ_{14} {added(bananas,1), added(oranges,2), switch(off), washed, ticked(oranges)}
μ_{15} {added(bananas,0), added(oranges,1), switch(off), washed}
μ_{16} {added(bananas,0), added(oranges,2), switch(off), washed}
μ_{17} {added(bananas,0), added(oranges,2), switch(off), washed, ticked(oranges)}

Figure 2.2: Transition system of `blenderAgent`. The knowledge base is, by definition, the same for all mental states. The contents of the belief base are listed for each mental state below the digraph. The the goal base is {filled, switch(on)} for all μ_i $(i \neq 8)$, and $\emptyset$ for μ_8.

Chapter 3

Model Checker

In the previous chapter, we used `blenderAgent` to illustrate and exemplify the GOAL agent language. Specifically, in Sect. 2.1, we introduced `blenderAgent` as an agent that washes pieces of fruit, puts them into the reservoir of a blender, and then switches the blender on. Additionally, we gave `blenderAgent`'s code in Fig. 2.1, and without further discussion assumed that code *correct*, i.e. implementing the behaviour that we described informally. When it comes to software, which has become notorious for their proneness to programming errors, such assumptions are naive. A mistake is quickly made, and, especially for safety-critical systems, the costs can be high (e.g. the Ariane 5 accident).

Of course, `blenderAgent` does not qualify as a safety-critical system. However, suppose we slightly refine `blenderAgent`'s hardware such that it no longer works in a kitchen with bananas and oranges, but serves at a pharmaceutical plant where it is responsible for mixing the right quantities of morphine and saline to make anaesthetics. Although `blenderAgent`'s code may remain the same, the stakes are much higher in its new working environment. That is, in the former case, a bug resulting in `blenderAgent` not adhering to the recipe results in distasteful juice. In the latter case, it can cost lives.

Thus, we want to determine whether `blenderAgent` is correct or not. What is required for this? Well, for one thing, we need a means of formally expressing `blenderAgent`'s desired behaviour. Examples of such *properties* in natural language are that "`blenderAgent` adheres to the recipe" and that "`blenderAgent` eventually believes to have switched on the blender". Unfortunately, natural language is too informal and ambiguous for our purpose; instead, we use a *temporal logic* instantiated for GOAL. Second, we need a procedure to determine whether `blenderAgent` adheres to such a property in temporal logic. The procedure of choice is *model checking*.

This chapter is organised as follows. In Sect. 3.1, we discuss temporal logic for GOAL. Subsequently, in Sect. 3.2, a treatise on model checking is given. Section 3.3 discusses the implementation of the model checking system that we implemented, and Sect. 3.4 concludes the chapter with a summary.

3.1 Temporal Logic

By the chapter's introduction, we need a formalism in which properties like "`blenderAgent` eventually believes to have switched on the blender" can be expressed. Temporal logic, a formalism to describe change over time, suits this purpose. Although various different temporal logics exist, we focus on *linear temporal logic* (LTL), introduced in [82]. At the end of this section, we briefly discuss alternatives.

3.1.1 Linear Temporal Logic

Both syntactically and semantically, LTL can be regarded as an extension of propositional logic with *temporal operators*. This is reflected in the syntax of an LTL *formula* as follows. An LTL

formula ϕ (or φ) is built from a set of atomic propositions $\mathcal{P}$, the boolean connectives, and the temporal operators X and U (e.g. [24]).

$$\phi \quad ::= \quad p \in \mathcal{P} \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{X}\phi \mid \phi\,\mathsf{U}\,\phi$$

We use square brackets to denote operator precedence. Additionally, we use the following abbreviations (e.g. [24]):

$$\begin{array}{lcllcl} \bot & = & p \wedge \neg p & \top & = & \neg\bot \\ \phi \vee \phi' & = & \neg[[\neg\phi] \wedge [\neg\phi']] & \phi \rightarrow \phi' & = & [\neg\phi] \vee \phi' \\ \mathsf{F}\phi & = & \top\,\mathsf{U}\,\phi & \mathsf{G}\phi & = & \neg\mathsf{F}\neg\phi \\ \phi\,\mathsf{R}\,\phi' & = & \neg[[\neg\phi]\,\mathsf{U}\,[\neg\phi']] & & & \end{array}$$

The set of all formulas that are *well-formed* according to the previous syntax is denoted by $\mathcal{L}_{\text{LTL}}$.

The X operator is commonly referred to as the *ne*X*t-time operator*, and alternatively denoted by N (cf. Next) or $\bigcirc$ in the literature (e.g. [100]). The F and G operators are commonly referred to as the F*inally operator*, and the G*lobally operator*. Alternative names for these operators are the *eventually operator* and the *always operator*; alternative denotations are $\Diamond$ and $\Box$ (e.g. [58]). Finally, U is called the U*ntil operator*, and R is called the R*elease* operator.

Informally, the semantics of LTL formulas is as follows. Purely propositional (sub)-formulas are interpreted *in a single state*, whereas formulas in which temporal operators occur are interpreted *over sequences of states*. We can think of the temporal operators as indicating in which states their arguments must be true. That is, a formula $\mathsf{X}\phi$ is true in the current state if its argument ϕ is true in the *next* state. An $\mathsf{F}\phi$ formula is true in the current state if its argument ϕ is true in some future state, i.e. *finally*. Conversely, a $\mathsf{G}\phi$ formula is true in the current state if its argument ϕ is true in all future states, i.e. *globally*. A formula $\phi\,\mathsf{U}\,\phi'$ is true if ϕ is true from the current state *until* some future state in which ϕ' is true. A formula $\phi\,\mathsf{R}\,\phi'$ can be read as *"ϕ releases ϕ' from the obligation of being true"*, and is true if ϕ' is true from the current state until ϕ is true in some future state; ϕ' must also be true in this state.

Later, we give a more formal account of the semantics of LTL formulas. First, we have a look at how we can combine LTL with Goal. Given our previous informal account of LTL's semantics, how can we express properties like "`blenderAgent` eventually believes to have switched on the blender"? Well, it is to be expected that we use the F operator to express the "eventually" part of this property. Subsequently, the argument of this F formula should be an expression that formalises the "believes to have switched on the blender" part. In fact, we already have a formalism for these kinds of expressions, namely mental state conditions. This motivates the instantiation of the set of atomic propositions $\mathcal{P}$ with the language of mental state conditions $\mathcal{L}_\Psi$, including those Mscs containing free Pl variables.

The latter may cause confusion as the first sentence of this subsection states that LTL is "an extension of propositional logic": the possible occurrence of free variables may seem contradictory to the fact that LTL is propositional. The reason that this is not the case is that the scope of the Pl variables that may occur in an Msc ψ is limited to ψ only: the free variables $free(\psi)$ in ψ do not exist outside ψ. As such, ψ is indeed a proposition from LTL's point of view. We revisit this matter later in the chapter (i.e. Sect. 3.3.1) when we treat implementation details. The previous property can now be expressed as F`bel(switch(on))`.

We proceed with a formal exposition of LTL's semantics when combined with Goal, provided $\mathcal{P} = \mathcal{L}_\Psi$. In general, the truth of an LTL formula is defined with respect to an infinite sequence of states. Applied to Goal, these infinite sequences are computations of an agent such that each state on the sequence is a mental state. Interpreting LTL formulas over the computations of a Goal agent allows us to express properties about its execution, i.e. its operational semantics as defined in Sect. 2.5. Let $\mathbf{\Pi}$ be the set of all computations of a Goal agent (see also Sect. 2.5), and let $\mathbb{N}$ be the set of natural numbers. The truth of an LTL formula is given by LTL's entailment relation $\models_{\text{LTL}}$ as follows (e.g. [24, 100]).

Definition 4. *Let* $\boldsymbol{\pi} \in \mathbf{\Pi}$*, let* $i \in \mathbb{N}^+$*.*
Then:

$$
\begin{array}{lll}
\boldsymbol{\pi}, i \models_{\text{LTL}} \psi & \text{iff} & \boldsymbol{\pi}_i \models_{\Psi} \psi \\
\boldsymbol{\pi}, i \models_{\text{LTL}} \neg\phi & \text{iff} & \boldsymbol{\pi}, i \not\models_{\text{LTL}} \phi \\
\boldsymbol{\pi}, i \models_{\text{LTL}} \phi \wedge \phi' & \text{iff} & \boldsymbol{\pi}, i \models_{\text{LTL}} \phi \text{ and } \boldsymbol{\pi}, i \models_{\text{LTL}} \phi' \\
\boldsymbol{\pi}, i \models_{\text{LTL}} \mathsf{X}\phi & \text{iff} & \boldsymbol{\pi}, i+1 \models_{\text{LTL}} \phi \\
\boldsymbol{\pi}, i \models_{\text{LTL}} \phi\,\mathsf{U}\,\phi' & \text{iff} & \text{there exists a } k \geq i \text{ such that :} \\
 & & \quad \boldsymbol{\pi}, j \models_{\text{LTL}} \phi \text{ for all } i \leq j < k, \text{ and } \boldsymbol{\pi}, k \models_{\text{LTL}} \phi'
\end{array}
$$

Note that for a $\phi\,\mathsf{U}\,\phi'$ formula to be true, ϕ' *must* become true eventually (provided ϕ is true until then). In contrast, a $\phi\,\mathsf{R}\,\phi'$ formula is true even if ϕ remains false forever (provided that ϕ' is always true). Due to this, the until and release operators are sometimes referred to as *strong until* and *weak release* [58]. We say that a computation $\boldsymbol{\pi}$ satisfies a property φ, denoted $\boldsymbol{\pi} \models_{\text{LTL}} \varphi$, if φ is satisfied by $\boldsymbol{\pi}$ at time index 0.

$$\boldsymbol{\pi} \models_{\text{LTL}} \varphi \quad \text{iff} \quad \boldsymbol{\pi}, 0 \models_{\text{LTL}} \varphi$$

We say that a GOAL agent P satisfies a property φ, denoted $P \models_{\text{LTL}} \varphi$, if all its computations do.

$$P \models_{\text{LTL}} \varphi \quad \text{iff} \quad \boldsymbol{\pi} \models_{\text{LTL}} \varphi \text{ for all } \boldsymbol{\pi} \in \boldsymbol{\Pi}$$

Like propositional formulas, LTL formulas can be rewritten to *negation normal form* (NNF) in which all negations are pushed inwards until they occur only before propositions. To do this, the following equivalences (e.g. [18, 24]) are applied inductively.

$$
\begin{array}{lcllcllcl}
\neg[\phi \wedge \phi'] & \equiv & [\neg\phi] \vee [\neg\phi'] & \neg\mathsf{X}\phi & \equiv & \mathsf{X}\neg\phi & & & \\
\neg[\phi \vee \phi'] & \equiv & [\neg\phi] \wedge [\neg\phi'] & \neg\mathsf{F}\phi & \equiv & \mathsf{G}\neg\phi & \neg[\phi\,\mathsf{U}\,\phi'] & \equiv & [\neg\phi]\,\mathsf{R}\,[\neg\phi'] \\
\neg[\phi \rightarrow \phi'] & \equiv & \phi \wedge [\neg\phi'] & \neg\mathsf{G}\phi & \equiv & \mathsf{F}\neg\phi & \neg[\phi\,\mathsf{R}\,\phi'] & \equiv & [\neg\phi]\,\mathsf{U}\,[\neg\phi']
\end{array}
$$

That is, from proposition logic, we know that $\wedge$ and $\vee$ are each other's *dual*. Additionally, the temporal operators F and G are each other's dual, and the same is true for U and R. In the remainder, all LTL formulas are assumed in NNF.

We will often refer to the LTL instantiation for GOAL discussed here as the GOAL *verification logic*. It should be noted that this logic has already been briefly introduced in [52]. We compare our verification logic with verification logics in the literature for other agent languages in the next chapter.

3.1.2 Other Temporal Logics

LTL is not the only temporal logic. For example, LTL is a subset of the more expressive logic CTL*, which in turn is a subset of the even more expressive μ *calculus* (e.g. [24]). Yet another temporal logic is *Hennessy-Milner logic* [50], in which actions play a more dominant role.

With respect to the model checking field, the most notable temporal logic besides LTL is *computation tree logic* (CTL), introduced in [20], which is also a subset of CTL* (yet different from LTL; see below). LTL and CTL differ in the way they treat the transition system of the program under investigation. Whereas LTL formulas are interpreted over all sequences of states that can be derived from the transition system individually, CTL formulas are interpreted over the *entire* transition system at once. To this end, CTL features additional operators, denoted A and E, that quantify universally and existentially over the paths that start in some current state of the system.

Although LTL and CTL have a common subset, there exist properties that can be expressed in LTL and not in CTL, and vice versa [99]. We give some examples. Suppose we want to verify the property "eventually, `blenderAgent` forever believes to have filled the blender". Let φ denote this property. In LTL, we express this as $\varphi = \mathsf{FG}\texttt{bel(filled)}$. In CTL, however, this cannot be express. Conversely, suppose the property under verification is "it is always the case that if the fruit is washed, then in some cases `blenderAgent` may believe to have filled the blender next, and in some cases not". Let φ' denote this property. In CTL, this can be expressed as

$\varphi' = \mathsf{AG}[\mathbf{bel}(\texttt{washed}) \rightarrow [\mathsf{EX}\,\mathbf{bel}(\texttt{filled}) \wedge \mathsf{EX}\neg\mathbf{bel}(\texttt{filled})]]$, whereas this cannot be expressed in LTL.

Whether one should use LTL or CTL as formalism to express properties in is one of the "great debates in model checking" (see Appx. B of [58] for an interesting overview of this and other "great debates"). The main reason *for us* to choose LTL is that it is claimed to provide a more natural way of thinking about programs. With respect to usability, user-friendliness, and learning curve of the model checking tool, this is very important. In a research paper written at IBM, the following is stated about CTL [86]:

> "We found only simple CTL equations to be intuitively comprehensible; nontrivial CTL equations are hard to understand and prone to error."

Other advantages of LTL are given in [99].

3.2 Model Checking

Having established a formalism to express properties in, we now look in more detail to the decision procedure for determining whether a Goal agent P satisfies an LTL specification φ. Our method of choice is *model checking*.

During the 1960s and 1970s, the prevailing method for establishing correctness of programs was by means of manual construction of mathematical proofs [21]. The construction of such proofs, however, is tedious, difficult, and requires human ingenuity: in practice, this renders manual verification unable to scale beyond small programs [21]. In an attempt to find a more practical approach to formal verification, in the early 1980s, model checking was proposed as a method for *automatically* establishing correctness of systems *without* the need for construction of elaborate proofs. The key insight that inspired model checking is that temporal logic can be used to reason about programs [82]. That is, given the transition system $\mathcal{T}$ of agent P with computations $\mathbf{\Pi}$ and an LTL specification φ, a model checker determines whether $\boldsymbol{\pi} \models_{\text{LTL}} \varphi$ for all $\boldsymbol{\pi} \in \mathbf{\Pi}$, i.e. whether $P \models_{\text{LTL}} \varphi$. If this is the case, then P is called a *model* of φ.

There are many approaches to the model checking problem. For LTL, we distinguish two main types: *tableau methods*, introduced in [70], and *automata-theoretic methods*, introduced in [101]. The latter have an important advantage over the former, namely that they can be applied under an *on-the-fly* regime: this means that transition system $\mathcal{T}$ is generated *during* rather than *before* the decision procedure (as is necessary in tableau methods). As $\mathcal{T}$ is typically very large, while often, only parts of it need be inspected to establish that it does *not* satisfy a property, benefits are gained: we do not need to spend resources on the generation and storage of parts of $\mathcal{T}$ beforehand, which are not inspected during the decision procedure later. Therefore, we apply an automata-theoretic method in this thesis.

The automata-theoretic approach to LTL model checking is based on the result that LTL formulas can be represented as *Büchi automata* [101]. The main difference between standard automata and Büchi automata (e.g. [94]) is that the latter *accept* words of infinite length, called *ω-words* (e.g. [94]), whereas the former's input is finite. How can this be used for model checking? Well, because we can construct a Büchi automaton corresponding to an LTL formula, we can construct such an automaton for the *negated* property under investigation, i.e. $\neg\varphi$. The intuition behind this is that instead of establishing that the Goal agent under verification P satisfies φ, we try to establish that all computations of P violate $\neg\varphi$. We call the automaton corresponding to $\neg\varphi$ the *property automaton*, and denote it by $\mathcal{A}_{\neg\varphi}$. Informally, all the infinite words that $\mathcal{A}_{\neg\varphi}$ accepts correspond to computations of some Goal agent that satisfy $\neg\varphi$, hence violate φ.

Additionally, we can construct a Büchi automaton that accepts words corresponding to computations (i.e. infinite sequences) of the Goal agent P. We call such an automaton a *program automaton*, and denote it by $\mathcal{A}_P$. Now, if the *language* of $\mathcal{A}_P$, denoted by $L(\mathcal{A}_P)$, and the language of $\mathcal{A}_{\neg\varphi}$, denoted by $L(\mathcal{A}_\varphi)$, are *not* disjoint, then there exists a word ω corresponding to a computation $\boldsymbol{\pi}$ that (i) violates φ, and (ii) is a computation of P. Then, by LTL's semantics, $P \not\models_{\text{LTL}} \varphi$, and $\boldsymbol{\pi}$ is called a *counterexample*.

To summarise the automata-theoretic approach, for the verification of an agent P with respect to a property φ we first represent P and $\neg\varphi$ as Büchi automata $\mathcal{A}_P$ and $\mathcal{A}_{\neg\varphi}$, and then determine whether the intersection of their languages is empty, i.e. whether $L(\mathcal{A}_P) \cap L(\mathcal{A}_{\neg\varphi}) = \emptyset$. If this is the case, then we conclude $P \models_{\text{LTL}} \varphi$, and otherwise $P \not\models_{\text{LTL}} \varphi$. To perform this check, a *product automaton* $\mathcal{A}_\times$ is constructed, whose language is the intersection of the languages of $\mathcal{A}_P$ and $\mathcal{A}_{\neg\varphi}$, i.e. $L(\mathcal{A}_\times) = L(\mathcal{A}_P) \cap L(\mathcal{A}_{\neg\varphi})$. As such, the automata-theoretic approach reformulates the model checking problem as a language emptiness problem for Büchi automata for which efficient algorithms exist (e.g. [92]). The procedure is shown algorithmically as Alg. 1.

A well-known automata-theoretic model checker, which we shall refer to later in this thesis as well, is SPIN (e.g. [58]): a model checker for the verification of concurrent imperative systems. The input language of SPIN is called PROMELA (acronym for *process meta language*; SPIN is an acronym for *simple* PROMELA *interpreter*), and is comparable to imperative languages like C. SPIN has for example [64] been used to verify correctness of the control systems of the movable storm surge barrier (named the Maeslantkering) near the Dutch harbour of Rotterdam (the Maeslantkering was built to prevent severe floods in the Netherlands, like in 1953, from happening).

We proceed as follows. First, we briefly give a more formal account of Büchi automata. Subsequently, we discuss considerations with respect to the check for language emptiness. We remark that the theory presented in this section is well-established in the model checking community, and repeated here for completeness and future reference. Details can be found in Appx. A, or in the referenced literature.

3.2.1 Büchi Automata

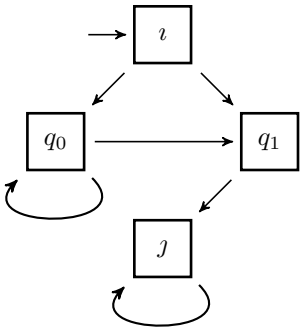
We commence with the formal definition of a Büchi automaton; basic familiarity with standard automata theory is assumed (e.g. [71]). In the model checking literature, many different notations are used to define Büchi automata. Here, we adopt the style of [89], in which a Büchi automaton $\mathcal{A}$ is defined as the tuple $\mathcal{A} = \langle \mathcal{Q}, \mathcal{I}, \delta, \mathcal{F}, \mathcal{D}, \Lambda \rangle$, in which:

- $\mathcal{Q}$ is a finite set of *states*,
- $\mathcal{I} \subseteq \mathcal{Q}$ is the set of *initial states*,
- $\delta : \mathcal{Q} \longrightarrow 2^{\mathcal{Q}}$ is the *transition function*,
- $\mathcal{F} \subseteq 2^{\mathcal{Q}}$ is a set of *acceptance conditions* $F \in \mathcal{F}$,
- $\mathcal{D}$ is a set of *labels*, called the *domain*, and
- $\Lambda : \mathcal{Q} \longrightarrow 2^{\mathcal{D}}$ is a *labelling function*.

In this definition, the domain $\mathcal{D}$ can be regarded as the equivalent of what is called the *alphabet* in standard automata theory, and the labels in $\mathcal{D}$ as the alphabet's *symbols*. Notice that labels

1: Construct a Büchi automaton $\mathcal{A}_P$ corresponding to P
2: Construct a Büchi automaton $\mathcal{A}_{\neg\varphi}$ corresponding to $\neg\varphi$
3: Construct a Büchi automaton $\mathcal{A}_\times$ such that $L(\mathcal{A}_\times) = L(\mathcal{A}_P) \cap L(\mathcal{A}_{\neg\varphi})$
4: **if** $L(\mathcal{A}_\times) = \emptyset$ **then**
5: **print** "$P \models_{\text{LTL}} \varphi$"
6: **else**
7: **print** "$P \not\models_{\text{LTL}} \varphi$"
8: **end if**

Algorithm 1: Let P be a GOAL program, and let φ be a property. Then: this algorithm checks whether $P \models_{\text{LTL}} \varphi$.



$$
\begin{aligned}
\mathcal{Q} &= \{\imath, q_0, q_1, \jmath\} \\
\mathcal{I} &= \{\imath\} \\
\delta(q) &= \begin{cases} \{q_0, q_1\} & \text{if } q \in \{\imath, q_0\} \\ \{\jmath\} & \text{if } q \in \{q_1, \jmath\} \end{cases} \\
\mathcal{F} &= \{\{\imath, q_1, \jmath\}\} \\
\mathcal{D} &= 2^{\mathcal{L}_\Psi} \\
\Lambda(q) &= \begin{cases} 2^{\mathcal{L}_\Psi} & \text{if } q \in \{\imath, q_0, \jmath\} \\ \{\Psi \in 2^{\mathcal{L}_\Psi} \mid \psi \in \Psi\} & \text{if } q = q_1 \end{cases}
\end{aligned}
$$

Figure 3.1: Büchi automaton for $\mathsf{F}\psi$.

are not associated with transitions, but with states by means of the labelling function Λ, and that states are labelled with a set of labels rather than with a single label. This was first observed in the LTL-to-Büchi translation algorithm presented in [44], and adopted by some of that algorithm's successors (including the algorithm that we apply).

We adopt the following terminology, notation, and definitions (e.g. [89]). A *run* of $\mathcal{A}$ is an infinite sequence of states $r = q_0 q_1 \cdots$ such that $q_0 \in \mathcal{I}$ and $\delta(q_i) = q_{i+1}$ for all $i > 0$. A run is called *accepting* if there exists a $q \in F$ that appears infinitely often on r for all $F \in \mathcal{F}$. A word $\omega = \lambda_0 \lambda_1 \cdots$ is an infinite sequence of labels from $\mathcal{D}$. A word is *accepted* by $\mathcal{A}$ if there exists an accepting run $r = q_0 q_1 \cdots$ such that $\lambda_i \in \Lambda(q_i)$ for all $0 \geq i$. Finally, the language $L(\mathcal{A})$ is the set of all words that are accepted by $\mathcal{A}$.

As an example, we treat the Büchi automaton for the LTL formula $\mathsf{F}\psi$, in which ψ is a mental state condition; the formal definition for generic LTL formulas is treated in Appx. A.2. The automaton is given in Fig. 3.1 both formally and graphically (to distinguish between transition systems and Büchi automata, we draw states of the former as circles, and states of the latter as boxes). It has four states of which $\imath$ is a *source state*, and $\jmath$ is a *sink state*. The source state is required to ensure that the automaton has only a single initial state (this is convenient for the algorithm that checks for language emptiness to be discussed in Sect. 3.2.2); the sink state is required to ensure that the automaton also accepts infinite runs corresponding to computations with a *finite* prefix on which truth of $\mathsf{F}\psi$ can be established. Details can be found in Appx. A.2.

From the source state, there are two transitions: one goes to q_0 itself, and the other goes to q_1. The same holds for q_0. In q_1, there is only one transition, namely to the sink state; the sink state only has a transition back to itself. All runs of this automaton start in the source state, followed by either an infinite sequence of q_0's, i.e. $\imath q_0 q_0 \cdots$, or a finite sequence of q_0s, followed by q_1, followed by an infinite sequence of $\jmath$'s, i.e. $\imath q_0 \cdots q_0 q_1 \jmath\jmath \cdots$.

The set of acceptance conditions $\mathcal{F}$ is a singleton containing one acceptance condition of three states $F = \{\imath, q_1, \jmath\}$. This means that at least one of these three states must occur infinitely often on all accepting runs. Thus, all runs of the automaton are accepting, except for the infinite sequence of q_0s, as q_0 is not a member of F (on all other runs, $\jmath \in F$ occurs infinite often). The domain $\mathcal{D}$ contains all possible subsets of mental state conditions (i.e. the power set of the language of MSCs), and the labelling function Λ assigns all these subsets as labels to the states of the automaton. An exception, however, is the labelling of q_1, which has the following constraint: its labels must contain ψ. Combined with the fact that all accepting runs are of the form $\imath q_0 \cdots q_0 q_1 \jmath\jmath \cdots$, all words ω accepted by the automaton are of the form $\omega = \lambda_0 \cdots \lambda_k \cdots$, in which k is some finite natural number, and $\lambda_k \in \Lambda(q_1)$ such that $\psi \in \lambda_k$.

The program automaton corresponding to a GOAL agent can be derived from its transition system (e.g. [93]), whereas the property automaton corresponding to an LTL property is constructed algorithmically. Various different translation algorithms exist; we chose the LTL2AUT algorithm [29], which improves the GPVW algorithm [44]. Although over ten years old, we favour LTL2AUT over more recent algorithms because of the following remark in [100]:

> "Two papers published in 1999 [namely [28, 29]] showed that [44] is not the last word on explicit LTL translation, which opened the door to many more papers. In fact, so many papers have been published over the last few years on this topic that it is difficult to say what is the best approach to translating LTL to automata."

Because for other translation algorithms it seems yet unclear whether they perform better, equal, or worse, we decided to apply their common predecessor LTL2AUT. In the implemented model checker, however, it is fairly easy to plug-in another translation algorithm (see also Sect. 3.3). That is, if future research establishes superiority of another algorithm, the necessary modifications to the current model checker are quickly made. The details about program and property automata (as well as examples featuring `blenderAgent`), can be found in Appxs. A.1 and A.2 (the latter also elaborates on the previous example). In the remainder, we denote a program automaton by $\mathcal{A}_P = \langle \mathcal{Q}_P, \mathcal{I}_P, \delta_P, \mathcal{F}_P, \mathcal{D}_P, \Lambda_P \rangle$ and a property automaton by $\mathcal{A}_{\neg\varphi} = \langle \mathcal{Q}_{\neg\varphi}, \mathcal{I}_{\neg\varphi}, \delta_{\neg\varphi}, \mathcal{F}_{\neg\varphi}, \mathcal{D}_{\neg\varphi}, \Lambda_{\neg\varphi} \rangle$ in which $\mathcal{D}_P = \mathcal{D}_{\neg\varphi} = 2^{\mathcal{L}_\Psi}$. Then, the product automaton $\mathcal{A}_\times = \langle \mathcal{Q}_\times, \mathcal{I}_\times, \delta_\times, \mathcal{F}_\times, \mathcal{D}_\times, \Lambda_\times \rangle$ such that $L(\mathcal{A}_\times) = L(\mathcal{A}_P) \cap L(\mathcal{A}_{\neg\varphi})$ is defined as follows (derived from [92]):

$$
\begin{array}{lcl}
\mathcal{Q}_\times & = & \mathcal{Q}_P \times \mathcal{Q}_{\neg\varphi} \\
\mathcal{I}_\times & = & \mathcal{I}_P \times \mathcal{I}_{\neg\varphi} \\
\\
\delta_\times(q_\times) & = & \begin{cases} \delta_P(q_P) \times \delta_{\neg\varphi}(q_{\neg\varphi}) & \text{if } q_\times = \langle q_P, q_{\neg\varphi} \rangle \\ & \text{and } \Lambda_P(q_P) \cap \Lambda_{\neg\varphi}(q_{\neg\varphi}) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\
\\
\mathcal{F}_\times & = & \widehat{\mathcal{F}_P} \cup \widehat{\mathcal{F}_{\neg\varphi}} \text{ in which } \widehat{\mathcal{F}_P} = \bigcup_{F_P \in \mathcal{F}_P} \{ F_P \times \mathcal{Q}_{\neg\varphi} \} \\
& & \text{and } \widehat{\mathcal{F}_{\neg\varphi}} = \bigcup_{F_{\neg\varphi} \in \mathcal{F}_{\neg\varphi}} \{ \mathcal{Q}_P \times F_{\neg\varphi} \} \\
\\
\mathcal{D}_\times & = & 2^{\mathcal{L}_\Psi} \\
\Lambda_\times(q_\times) & = & \Lambda_P(q_P) \cap \Lambda_{\neg\varphi}(q_{\neg\varphi}) \text{ in which } q_\times = \langle q_P, q_{\neg\varphi} \rangle
\end{array}
$$

Again, we refer to Appx. A.3 for the details and an example of this construction.

3.2.2 Checking Language Emptiness

We proceed with some considerations with respect to algorithms that, given a Büchi automaton $\mathcal{A} = \langle \mathcal{Q}, \mathcal{I}, \delta, \mathcal{F}, \mathcal{D}, \Lambda \rangle$, establish whether $\mathcal{A}$'s language is empty or not. Typical use of such algorithms in this thesis is with the product automaton corresponding to a GOAL agent P and a negated property $\neg\varphi$.

There are three types of algorithms for establishing the non-emptiness of the language of a Büchi automaton: *explicit-state methods* (e.g. [27]), *symbolic methods* (e.g. [74]), and SAT-*based methods* (e.g. [8]). Because the relative merits of the latter approach seem at current unclear [100], we favour an explicit-state or symbolic approach. Although neither of these two methods is necessarily superior to the other [58], explicit-state methods are favoured for software model checking, whereas symbolic methods tend to perform better at hardware verification [36]. As GOAL agents are software, we chose an explicit-state approach.

Within the class of explicit-state algorithms, two types can be distinguished: those based on the computation of *maximal strongly connected components* (MSCC) in the automaton's state space, and those based on a *nested depth-first search* (NDFS) through the automaton's state space. The MSCC algorithms have as a drawback that they require the state space of the entire automaton to be in main memory. As product automata are typically very large, this can pose problems in practice [100]. In contrast, NDFS algorithms do not require the entire state space to be stored (see for example [47] for an example in which no states are stored at all, i.e. a *state-less search*) such that they can tackle larger model checking problems.

The first Ndfs algorithm was proposed in [27]. Various modifications and improvements to that algorithm have been proposed over the years. We have chosen for the Ndfs algorithm of [90, 91], which is claimed the best of its type in [100]. The algorithm is given and described in Appx. A.4. An important characteristic of this algorithm is that it can be applied under an on-the-fly regime such that the product automaton is constructed during the depth-first search rather than before it. If the Ndfs algorithm establishes that the language of the automaton is non-empty before the entire automaton is constructed then we have not spent resources on the construction of parts of the automaton that are not visited during the search.

3.2.3 Performance Issues

Model checking has, like manual proof construction, issues of its own. The major problem is called *state space explosion* (e.g. [21]), and is caused by the enormous number of states that the transition system of a program may contain. As a result, the program and product automata are very large such that the emptiness check requires many resources, both in terms of time and space. On the one hand, this means that model checking algorithms need be implemented as efficient and with as little overhead as possible. On the other hand, *state space reduction techniques* can be developed to prune the transition system without affecting the model checker's results. In this part of the thesis, we will not discuss reduction techniques yet; the second half of the thesis is devoted to those.

3.3 Implementation

In the previous, we have discussed the specification language in which properties about Goal agents can be specified, namely LTL for Goal, and the theory of how we can algorithmically establish whether such a property is satisfied, namely by means of *on-the-fly* Ndfs *explicit-state automata-theoretic LTL model checking*. Based on these theories, we have developed a Goal-specific model checker by implementing all automata-theoretic algorithms from scratch *using the standard* Goal *interpreter*. This approach is fundamentally different from past efforts in the field of model checking of agent programs, because those efforts all rely on an existing model checker for at least the check for language emptiness of the product automaton. Also, they do not use the standard interpreter of an Apl for generation of a program automaton, but a custom-built interpreter for the sake of model checking alone (that often offers less functionality). We elaborate on these differences as well as on the advantages of our approach in detail in the next chapter; here, we focus on the implementation of our *interpreter-based model checker* in Java. We start with a general discussion on the implementation's architecture, and proceed with the details of the use of the Goal interpreter.

3.3.1 Architecture

Although we aimed at implementing the model checking algorithms specifically for Goal, we ended up with a generic Java model checking framework with independent components that can be plugged-in and plugged-out easily. As a result, the implemented framework facilitates model checking of *any* language with an operational semantics. We acknowledge that generic frameworks are not always the best solution to a given problem (e.g. [107]). In our case, however, we deemed it necessary that our system would be highly flexible. The reason is as follows.

From the previous section, it is apparent that when developing a model checker, there are *many* degrees of freedom. As a consequence, we needed to make design decisions during the early stages of development of which we were not yet completely sure whether they would turn out to be the best choices possible. A striking example of this is the choice of LTL-to-Büchi translation algorithm (recall that at current, there is no consensus in the model checking community about which algorithm is best). If superiority of some LTL-to-Büchi algorithm is established in the future, we want to be able to remove the Ltl2Aut algorithm from our current implementation,

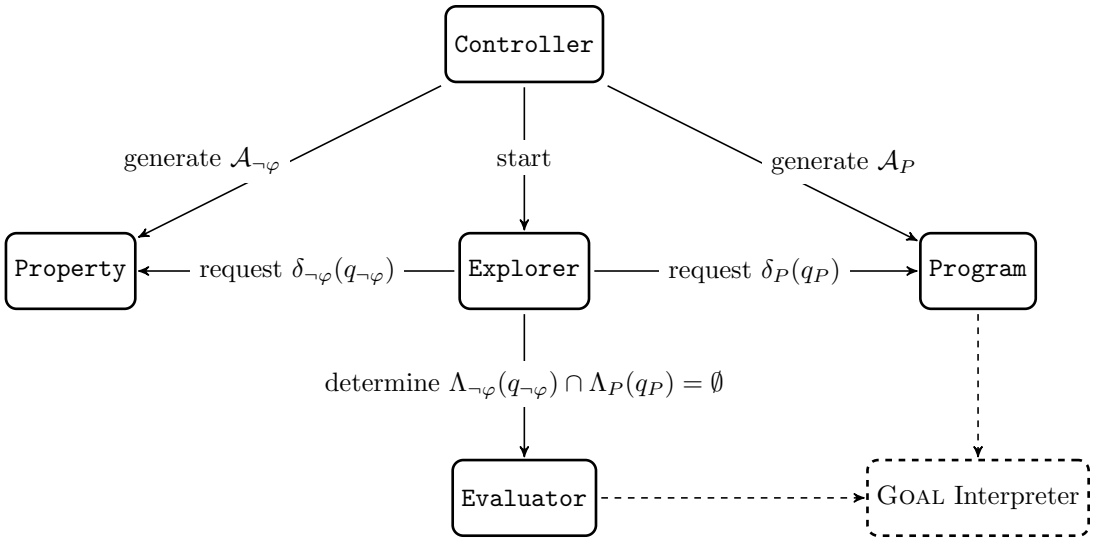


Figure 3.2: Architecture of the model checker. Dashed elements are GOAL-specific.

and incorporate the superior one without the need to re-implement the entire system. Thus, a modular design is a necessity.

To this end, the architecture of our model checking system (shown graphically in Fig. 3.2) consists of components that can be implemented independently, have distinct responsibilities, and communicate with each other strictly through well-defined interfaces such that each component can easily be plugged-in (and out) without affecting the other components. In what follows, we treat these different components and illustrate their use with a description of their current implementation for GOAL.

Controller component

The *controller component*, denoted `Controller`, is only responsible for everything that need be done *before* the product automaton $\mathcal{A}_\times$ is checked for emptiness; once this check commences, `Controller` is done. During this initial phase of the model checker's execution, `Controller` processes the user-input (i.e. the code of a GOAL agent and the property under investigation as strings of text) to a more convenient representation. Also, depending on whether additional directives are provided, it may decide to issue requests to other components (i.e. `Program` and `Property`) for pre-computation of the entire program and property automata *before* exploration of the product automaton. This corresponds to the decision procedure as outlined in Alg. 1, but renders the execution of the system *not-on-the-fly*. Because on-the-fly construction of the automata is favoured, the default of the current implementation is that `Controller` does not request pre-computation. We come back to on-the-fly-ness later. At the end of the initial phase, `Controller` signals `Explorer` that it may start the check for emptiness.

Explorer component (using [91])

The *explorer component*, denoted `Explorer`, is the core component of the system. Its task is to check the product automaton for emptiness, and does so by exploring its state space with the NDFS algorithm of [91] (mentioned previously, and treated in detail in Appx. A.4) with the following minor modifications. The reason is that the algorithm in [91] assumes the entire $\mathcal{A}_\times$ to be generated, while we apply an on-the-fly exploration regime, in which $\mathcal{A}_\times$ is generated as needed, starting from its initial state $\langle \imath_P, \imath_{\neg\varphi} \rangle$. This requires some additional bookkeeping, most of which is quite straightforward. Interesting is, however, the way `Explorer` deals with the (partial) definition of the transition function. We start our explanation of how this works by remarking that

`Explorer` is called "explorer component" for a reason: it *explores*. We emphasise this, because `Explorer` is *not* responsible for the *generation* of either automata. Instead, it delegates this task to other components (namely `Program` and `Property`). The same holds for checking whether the sets of labels on the states q_P and $q_{\neg\varphi}$ in some product state $q_\times = \langle q_P, q_{\neg\varphi}\rangle$ are disjoint; this is delegated to `Evaluator`.

In the current implementation, this works as follows. During a verification run, the transition function $\delta_\times$ need be computed for some $q_\times = \langle q_P, q_{\neg\varphi}\rangle$ at two places in the [91] algorithm (line 6 and line 23 of Algs. 7 and 8 in Appx. A.4). As soon as this need arises, `Explorer` "unpacks" $q_\times$, and requests `Program` for the successors of q_P in the program automaton $\mathcal{A}_P$ to which q_P belongs. Similarly, `Explorer` requests `Property` for the successors states of $q_{\neg\varphi}$. Once all these successors are returned to `Explorer`, it computes their product, which are exactly the successors of $q_\times$ (by definition of the product automaton). Because only product states whose constituents q_P and $q_{\neg\varphi}$ have a label in common are actually granted successors (also by definition of the product automaton), `Explorer` first asks `Evaluator` whether a common label in $\Lambda_P(q_P)$ and $\Lambda_{\neg\varphi}(q_{\neg\varphi})$ exists, before issuing requests for the automata-specific successors of q_P and $q_{\neg\varphi}$ to `Program` and `Property`. If this is the case, then `Evaluator` returns "true", and "false" otherwise.

As a result of the above sketched procedure, the product automaton is *always* constructed on-the-fly. Indeed, the states of the product automaton are generated incrementally starting from the initial state $\langle \imath_P, \imath_{\neg\varphi}\rangle$. However, this says nothing about the on-the-fly-ness of the entire decision procedure: if `Controller` has already requested `Program` and/or `Property` to compute the entire program and property automata *before* it asks `Explorer` to start the emptiness check, the decision procedure cannot really be called on-the-fly, because both automata have already been generated. Note also that because `Explorer` treats `Program` and `Property` as black boxes, it does neither know nor care about the actual on-the-fly-ness of the decision procedure: `Explorer` is only interested in receiving successors, and not in how `Program` and `Property` obtain these.

Property component (using Ltl2Aut)

The *property component*, denoted `Property`, can receive requests from both `Controller` and `Explorer` (see Fig. 3.2). The former can ask `Property` to generate the entire Büchi automaton corresponding to the negated property under investigation at once, whereas `Explorer` requests `Property` for the successors of a single state $q_{\neg\varphi}$. If, upon a request of `Explorer`, the property is already generated because of an earlier request of `Controller`, then `Property` can simply fetch the required successors from memory. Otherwise, it needs to compute the successors on-the-fly by applying the algorithm in use (currently Ltl2Aut).

Note that `Property` is strictly a *server*: it responds to requests from `Controller` and `Explorer`, but does not issue requests to other components itself.

Program component (using the Goal interpreter)

The *program component*, denoted `Program`, is quite similar to the property component. That is, it has the same interface, is strictly a server, and is treated no different by `Controller` and `Explorer`. In fact, from a theoretical point of view, the only difference is that it represents just another Büchi automaton. In practice, however, there is an important difference in the way that the automata are constructed. Whereas `Property` constructs $\mathcal{A}_{\neg\varphi}$ *algorithmically* (by means of Ltl2Aut), the program automaton $\mathcal{A}_P$ is constructed by `Program` *interpretatively*.

This works as follows. Any programming or modelling language with semantics defined as a transition system (i.e operational semantics) can by implemented by means of an interpreter; this interpreter can be used to generate the transition system of a program written in the respective language. As `Program` is responsible for the computation of $\mathcal{A}_P$ (either entirely at once or incrementally), which is defined by means of the transitions system of the program under investigation, it needs to generate this transition system. Hence, we may regard `Program` as a language interpreter.

Rather than implementing `Program` by writing a new interpreter for GOAL, we implemented `Program` as a "bridge" between the model checking system and GOAL's standard JAVA/PL interpreter. That is, upon a request for successors of some state q_P by `Explorer` (or a request for full generation by `Controller`), `Program` feeds q_P, which is a mental state μ, to the interpreter. The interpreter then starts computing μ's successors, and returns them to `Program`. As such, the interpreter *completely* takes care of the implementation of the semantics of the language (in this case GOAL), which is one of the advantages of our interpreter-based approach over previous approaches to APL model checking (see also Chap. 4). We treat the details of the interaction between `Program` and the GOAL interpreter in the next subsection.

Evaluator component (using GOAL)

Finally, the *evaluator component*, denoted `Evaluator`, takes care of label comparison. The purpose of the comparison of labels is to determine whether a program state q_P (which is a mental state by definition of the program automaton) does not violate the propositional part of a property state $q_{\neg\varphi}$. In practice, this is easier checked by evaluating the respective mental state conditions in q_P (recall that propositions are mental states when model checking GOAL) rather than by explicit computation of the labellings. That is, if all positive mental state conditions in $q_{\neg\varphi}$ are true in q_P, and all negative mental state conditions in $q_{\neg\varphi}$ are false in q_P, then there must exist a common corresponding label in $\Lambda_P(q_P)$ and $\Lambda_{\neg\varphi}(q_{\neg\varphi})$; the existence of such a label is all we need to know.

To evaluate an MSC in a mental state, we again use the GOAL interpreter, which already has an MSC querying mechanism (for determining applicability of action rules). Consequently, *all* MSCs that the interpreter can handle are allowed as proposition in LTL, including those containing PL variables or compound PL queries, and those referring to rules in the knowledge base. We stress this for two reasons. First, it is unusual in agent verification that verification logics are this expressive (see Sect. 4.1). Second, we place an additional remark on the occurrence of PL variables in LTL formulas (see also Sect. 3.1.1). Although the possibility to incorporate these in property specifications makes the verification logic quite powerful, there is also a danger with respect to human interpretation. Consider, for example, the following formula:

$$\phi \quad = \quad \mathsf{G}[\mathbf{bel}(\texttt{ticked(F)}) \rightarrow \mathbf{bel}(\texttt{recipe(F,Q),added(F,Q)})]$$

One might be tempted to think that this formula expresses that always if `blenderAgent` believes to have ticked the check box corresponding to `F`, then it also believes to have added sufficiently many pieces of *the same* `F`. This is, however, not what φ expresses, because the scope of `F` is limited to the mental state condition in which it occurs. For example, suppose μ is a mental state such that $\Sigma = \{\texttt{ticked(bananas)}, \texttt{added(oranges,2)}\}$ and $\texttt{recipe(oranges,2)} \in K$. Then, `F` can be bound to `bananas` during evaluation of $\mathbf{bel}(\texttt{ticked(F)})$, making it true. Subsequently, during the evaluation of $\mathbf{bel}(\texttt{recipe(F,Q),added(F,Q)})$, `F` can be bound to `oranges` and `Q` to `2`, making this MSC true as well. Consequently, ϕ is not violated in μ, which may seem unintuitive. To prevent possible confusion, we deem it good practice to use different identifiers for variables in different mental state conditions occurring in the same LTL formula, e.g.:

$$\phi' \quad = \quad \mathsf{G}[\mathbf{bel}(\texttt{ticked(F1)}) \rightarrow \mathbf{bel}(\texttt{recipe(F2,Q),added(F2,Q)})]$$

3.3.2 State Generation using the GOAL Interpreter

In the previous, we outlined how the program component of the architecture is used as a bridge between the exploration component and the GOAL interpreter. Our statement that we simply feed the interpreter μ and get *all* successors in return without any effort was, however, a little too optimistic: although we can request the GOAL interpreter for all action options given a mental state μ, only one of these options can be actually performed. Consequently, only one successor instead of all of them is obtained. The reason is that when performing an action, the JAVA objects and PL databases that represent the mental state in the standard GOAL interpreter are *updated*

rather than *replaced* with a new Java/Pl structure. Hence, as soon as we perform one action of the set of all options the "current" mental state is lost, because it is transformed to its successor.

To resolve this, we have tried several different solutions. With the most straightforward method, which we call *explicit-state cloning*, the entire Java/PL structure corresponding to a mental state is cloned explicitly, i.e. for every Java object in the one structure, an identical Java object in the other is created, and likewise for Pl databases and their contents. We distinguish two variants, *post-cloning with reversion* and *pre-cloning*, and already remark before their description that they are only relevant from a historical perspective; a better solution is treated later.

post-cloning with reversion Let μ be some mental state, and suppose there are n action options in μ (these action options can be obtained with a single method invocation to the interpreter). Then, for all $0 \leq i < n$, the program component: (i) performs option α_i in μ resulting in the successor μ'_i, (ii) clones μ'_i, and (iii) *reverses* the effects of α_i such that μ is retrieved. The last step is crucial, because otherwise, option α_{i+1} is performed in the wrong mental state. At the end of this process, n new Java/Pl structures exist that correspond to μ's n successors.

One of the drawbacks of this method is that we need a means to reverse actions; this is not natively supported by the interpreter. Thus, we needed to add this functionality to it, which is undesirable for various reasons. For one thing, it extends the code base of the interpreter with code that is unnecessary for interpretation of agents. Nevertheless, the respective methods need be maintained. Another reason is that we want to keep the connection between the program component and the interpreter as loose as possible for the sake of code maintainability and modularity. That is, the lesser invocations to methods of the interpreter, the better. Finally, implementing reversal of actions is something that concerns the semantics of Goal, while one of the advantages of our interpreter-based approach should be that we get the implementation of the semantics of the language under consideration "for free".

pre-cloning A better explicit-state cloning method is the following. Rather than first performing an action option α_i in μ and then cloning μ'_i, we first clone μ n times, yielding $\mu_0, \ldots, \mu_{n-1}$, and then perform α_i in μ_i. As such, we circumvent the need for reversions. Not only does this resolve the aforementioned issues, but it is also a faster method, because we do not need to spend time on reversing actions.

Unfortunately, after implementing both methods, we had to conclude that the explicit-state cloning approach is not a viable solution. The reason is that creating a clone of an entire Java/Pl structure corresponding to a mental state is simply too resource consuming: on the one hand, the cloning process takes a lot of time, and on the other hand, the memory demands of storing all these structures are very high. Additionally, comparing these structures for equality (a frequent operation in model checking to determine whether a state has already been visited) is also too time-consuming.

Our second solution, which we call *symbolic cloning*, has turned out to be *much* more efficient. The idea is to represent mental states symbolically as bit strings, called *binary mental states*, in which each bit corresponds to a belief or goal.

Binary mental states

The basic theory of binary mental states is quite straightforward. Let $index : (\mathcal{L} \cup 2^{\mathcal{L}}) \longrightarrow \mathbb{N}$ be a partial injective function from Pl facts (i.e. beliefs) and Pl sets of facts (i.e. goals) to the natural numbers. Then, the binary representation $\mathcal{B}(\mu)$ of a mental state $\mu = \langle K, \Sigma, \Gamma \rangle$ is defined as a sequence of bits.

$$\mathcal{B}(\mu) \quad = \quad b_0 b_1 \cdots \text{ in which } b_i = \begin{cases} 1 & \text{if there exists a } \chi \in \Sigma \text{ such that } index(\chi) = i \\ & \text{or there exists a } \gamma \in \Gamma \text{ such that } index(\gamma) = i \\ 0 & \text{otherwise} \end{cases}$$

Note that the knowledge base is not considered in the translation. The reason is that knowledge is static, hence we do not need to store it in each binary mental state. As an example, we consider the first 4 mental states of `blenderAgent` (see Fig. 2.2). Let *index* define *at least* the following mappings.

$$\begin{array}{lcllcl} \mathit{index}(\texttt{added(bananas,0)}) & = & 0 & \mathit{index}(\texttt{washed}) & = & 4 \\ \mathit{index}(\texttt{added(oranges,0)}) & = & 1 & \mathit{index}(\texttt{added(oranges,0)}) & = & 5 \\ \mathit{index}(\texttt{switch(off)}) & = & 2 & \mathit{index}(\texttt{added(bananas,1)}) & = & 6 \\ \mathit{index}(\texttt{filled, switch(on)}) & = & 3 & \mathit{index}(\texttt{added(bananas,2)}) & = & 7 \end{array}$$

Then, we have the following:

$$\mathcal{B}(\mu_0) = 1111\cdots \quad \mathcal{B}(\mu_1) = 011111\cdots \quad \mathcal{B}(\mu_2) = 0111101\cdots \quad \mathcal{B}(\mu_3) = 01111001\cdots$$

In the above, "$\cdots$" represents an infinite postfix of 0-s

Binary mental states require much less memory to store (the infinite postfix of 0-s is not stored), can be compared for equality with each other very quickly (using bitwise arithmetic), and are easy to clone. There is only one problem: the function *index*. In practice, it is impossible to define *index* before model checking commences, because from the code of a GOAL agent P alone, it cannot be established which beliefs and goals P *exactly* will have at runtime; only the beliefs and goals in the initial mental state are known. To resolve this, we start the verification run with *index* defined for the initial beliefs and goals only, and *dynamically* track "un-indexed" beliefs and goals during the verification run. Indices are assigned to beliefs and goals on a first-come-first-serve basis, i.e. in the order in which they occur. Let k be the number of beliefs and goals in the initial mental state. Then, these initial beliefs and goals receive indices ranging from 0 up to and including $k-1$ (see also the first column of the definition of *index* in the previous example). To illustrate how indices are assigned for beliefs and goals that are encountered during verification, we continue the example. When the fifth mental state μ_4 is encountered by the model checker, the program component will detect that the belief `ticked(bananas)` has not been indexed yet. At that moment, the last index assigned is 7 such that `ticked(bananas)` becomes associated with 8. The binary representation of μ_4 is then $\mathcal{B}(\mu_4) = 011110011\cdots$.

The implementation of this dynamic tracking requires quite the administration, because we want to do it efficiently. For example, determining whether a belief or goal has already been indexed can be done by iterating over all indexed beliefs and goals, but this would require time linear in the number of these indexed beliefs and goals. Instead, we use hashing data structures such that these checks can be done in constant time.

To the best of our knowledge, representing the state of an agent as a bit string has not been done before in agent model checking.

Symbolic cloning

With symbolic cloning, throughout the entire system, mental states occur in their binary form as described above. However, the GOAL interpreter cannot handle this binary representation to generate successors, i.e. it needs the explicit JAVA/PL structure corresponding to a binary mental state. To this end, the program component maintains a single such structure at runtime, denoted `m` in the remainder, that is repeatedly *updated* to accord with a specific binary mental state. Let $\beta = \mathcal{B}(\mu)$ be the binary representation of mental state μ. Then, we write $\texttt{m} \sim \beta$, $\texttt{m} \sim \mathcal{B}(\mu)$, or $\texttt{m} \sim \mu$ to denote that the JAVA/PL structure `m` corresponds to μ. Successor generation then works according to Alg. 2.

Upon a request for successors of some binary mental state $\beta = \mathcal{B}(\mu)$ corresponding to a mental state μ, first an empty set of successors is created (line 1). The following step (line 2) of the algorithm is to update `m` such that it corresponds to β. We treat how this works shortly. On line 3, the interpreter is requested for the action options in μ by feeding it `m`. The loop on lines 4-8 computes the successors as binary mental states as follows. On line 5, an option α is performed in `m` by requesting the interpreter to do so. As a result, `m` is transformed such that after the

invocation, $\mathtt{m}$ corresponds to a successor $\mu' = \mathcal{M}(\alpha, \mu)$ of μ rather than to μ itself. By computing the binary representation of $\mathtt{m}$ after performing α in it, thus, we obtain this μ' in its binary form. On line 6, this binary mental state is added to the set of successors. Finally (on line 7), we need to update $\mathtt{m}$ such that it corresponds to μ in the next iteration of the loop.

In practice, to update $\mathtt{m}$ such that it corresponds to a binary mental state $\beta = \mathcal{B}(\mu)$, we first translate $\mathtt{m}$ to its binary form $\mathcal{B}(\mathtt{m})$, and then compare bits in β and $\mathcal{B}(\mathtt{m})$ pairwise. Let β_i and $\mathcal{B}(\mathtt{m})_i$ be the i-th bit in the respective sequence. The following cases are possible.

$\beta_i = 0, \mathcal{B}(\mathtt{m})_i = 0$ In this case, the belief or goal that has assigned index i is neither in μ, nor is it in $\mathtt{m}$. Thus, we can leave $\mathtt{m}$ as it is with respect to this bit.

$\beta_i = 0, \mathcal{B}(\mathtt{m})_i = 1$ In this case, the belief or goal that has assigned index i is not in μ, but it is in $\mathtt{m}$. Thus, we need to remove this belief or goal from $\mathtt{m}$.

$\beta_i = 1, \mathcal{B}(\mathtt{m})_i = 0$ In this case, the belief or goal that has assigned index i is in μ, but it is not in $\mathtt{m}$. Thus, we need to add this belief or goal to $\mathtt{m}$.

$\beta_i = 1, \mathcal{B}(\mathtt{m})_i = 1$ In this case, the belief or goal that has assigned index i is in μ, and also in $\mathtt{m}$. Thus, we can leave $\mathtt{m}$ as it is with respect to this bit.

To give an indication of the space reduction, we briefly discuss the more specific issue related to storing beliefs (i.e. ground facts) only. Suppose that for explicit representation of beliefs we would use a string representation. A reasonable measure of size needed in that case would be the length of that representation, i.e. string length (in terms of bytes). For the sake of argument, let us assume that we can approximate space requirements by the average length of such strings, and that on average we would need L bytes to store a belief (this is an underestimate of what is really needed). Moreover, suppose we have N different beliefs which may or may not occur in a belief base. This yields 2^N possible belief bases. The average number of beliefs in these belief bases is $\sum_0^N k \cdot \binom{N}{k}$ divided by 2^N, which equals $N/2$. Explicit-state representation (using only belief bases) would thus require $N \cdot 2^{N-1} \cdot L$ bytes, or $N \cdot 2^N \cdot 4 \cdot L$ bits. Instead, the binary representation would require $N \cdot L$ bytes to represent the *index* mapping (negligible), and $N \cdot 2^N$ bits to represent the state space. This is a conservative estimate of the space reduction, which shows that minimally space requirements are reduced with a factor $4 \cdot L$.

State representation in the literature

The symbolic cloning approach described above exists by virtue of the binary representation of mental states. Representing states of the program under investigation efficiently is a challenge for all software model checkers. Here, we have a look at how other software verification tools deal with this.

We start with VeriSoft, presented in [47], which is the first model checker for verification of programs written in an actual programming language. Specifically, VeriSoft is a model checker

---

1: $successors := \emptyset$
2: update $\mathtt{m}$ such that it corresponds to β (i.e. such that $\mathtt{m} \sim \mu$)
3: $Act :=$ the set of action options in μ (computed by feeding $\mathtt{m} \sim \mu$ to the interpreter)
4: **for all** $\alpha \in Act$ **do**
5: perform α in $\mathtt{m}$ such that $\mathtt{m} \sim \mathcal{M}(\alpha, \mu)$
6: $successors := successors \cup \{ \mathcal{B}(\mathtt{m}) \}$
7: update $\mathtt{m}$ such that it corresponds to β
8: **end for**

Algorithm 2: Let $\beta = \mathcal{B}(\mu)$ be the binary representation of a mental state μ, and let $\mathtt{m}$ be the Java/Pl structure for a mental state. Then: this algorithm computes the successors of μ in binary form.

for verification of C/C++ programs, and does not store any states at all: C/C++ program's states are deemed too complex to be encoded efficiently. Instead, another mechanism, based on *sleep sets*, is used to keep track of states that have already been encountered during the search. Sleep set methods (e.g. [46]) were originally invented as a means to alleviate the state explosion problem discussed in Sect. 3.2.3; we treat sleep sets in that context in more detail in Chap. 7. Obviously, our approach differs from the VeriSoft approach, as we do store states (in their binary form, that is).

The necessity of representing states efficiently when model checking software is confirmed in [69], in which it is stated that even for relatively simple Java programs, more than 2 KB of memory is required for the representation of a single state. The same paper presents a more efficient representation that is used in the Java model checker Jpf [103] (acronym for Java *pathfinder*). This solution is based on the *collapse method*, originally introduced in [56], and tries to exploit the observation that when a new state is generated, large parts of the state are unchanged. Specifically, the collapse method tries to prevent these unchanged parts from being stored again. It does so by associating each component of the system (e.g. a Java class field) with an index. Then, a state can be *collapsed* to a list of indices, indicating which components comprise the state. Now, if a component is unchanged between two successive states, then the same index can be used to characterise both states with respect to that component, without storing two duplicates of it explicitly.

There appear similarities between the collapse method and our binary mental states. That is, we also associate each "component" (in our case beliefs and goals) with indices, and represent a mental state based on these indices. However, whereas in Jpf's collapse method a state is a vector of integers (serving as indices), we represent a mental state as a bit string. Another difference is the origin of the two methods. Whereas the collapse method was designed to prevent the storage of duplicate information about states, binary mental states were in the first place introduced to resolve our cloning issues.

Finally, we spend a few words on Spin's famous *bitstate hashing* technique (also known as *supertrace*). With this method, introduced in [55], rather than the states themselves only their *hash codes* are stored. This greatly reduces both verification time and memory consumption at the cost of impreciseness: if two states are mapped to the same hash code, then as soon as one of them has been processed (and is marked as such by storing its hash value) the other state will be thought of as processed as well. That is, the moment this other state is actually encountered for the first time, the model checker thinks it has already processed it before, and will not reconsider it, potentially missing a behaviour of the program that might violate the property. Thus, with bitstate hashing, *false negatives* may occur. To minimise such hash collisions, the specific computation of hash codes is crucial when bitstate hashing is applied (there are various techniques, e.g. using multiple different hash functions [58]).

With respect to our binary mental state approach, we may interpret the bit string representation of a mental state as a numerical value, which can serve as a hash code. Under this interpretation, our binary mental state approach can be seen as an application of bitstate hashing to Goal. An important difference, however, is that we must not only be able to convert mental states to their binary form (i.e. compute the hash code of a mental state), but also to convert a binary mental state back to its explicit form (i.e. retrieve the mental state from a hash code). This latter is not possible with bitstate hashing as implemented in Spin.

3.4 Summary

In this chapter, we have discussed the theory and implementation of a novel model checker for Goal that is built on top of the standard interpreter. In the first two sections, we have presented a verification logic for Goal based on LTL, and briefly discussed the theory of model checking and considerations for choosing particular algorithms and approaches (for the details of the specific theories we referred to Appx. A). With respect to implementation, we have presented the architecture of the implemented system: a model checking framework for any language with an

operational semantics, in this thesis instantiated for Goal. Additionally, we discussed the interaction between this framework and the Goal interpreter, which is one of the novelties of our approach to agent language model checking. A keystone of this interaction is the conversion of mental states to their binary representation and back.

Chapter 4

Comparison

Having presented our model checking system, in this chapter, we have a look at how it compares to existing model checkers in agent verification. In particular, we present a quantitative performance comparison of our model checker and other model checkers for the verification of GOAL agents, as performance is the main issue with model checking. An abridged version of this and the previous chapter has been published as [63].

The chapter is organised as follows. In Sect. 4.1, we give a brief overview of work that is closest to ours in the field of agent verification. Section 4.2 treats considerations concerning the design of our quantitative comparison. Subsequently, in Sect. 4.3, we treat the experiments and results, followed by a discussion in Sect. 4.4. The chapter is concluded with a summary in Sect. 4.5.

4.1 Program Model Checking in Agent Verification

The work presented in this thesis is not the first in the field of agent model checking. We identify two lines of research within the area: the one focusses on the development of tools for the verification of *models* of (multi-)agent systems, whereas the other focusses on developing verification tools for actual implementations, i.e. *programs*, of (multi-)agent systems. As the work presented in this thesis belongs to the second category, which we term APL *model checking* in the remainder, we give a brief overview of the existing literature in this sub-area. After that, we discuss *how*, and moreover, *why* our approach is different. For an overview of the other line of research, we refer to Sect. 2.3 of [83].

Although the first theoretical investigations on model checking of (multi-)agent systems have been published in the late 1990s [5, 6, 7], to the best of our knowledge, the first practice-oriented record on APL model checking is the 2002 paper [105] (later extended and republished as [106]). In this paper, programs written in the agent language MABLE are translated to PROMELA and verified with SPIN. In the same time period, a number of papers on model checking AGENTSPEAK programs have been published, namely [11, 10, 14, 12, 13]. Like the papers on model checking MABLE the AGENTSPEAK efforts are based on a translation from the AGENTSPEAK code to either PROMELA (combined with the SPIN model checker) or JAVA. In the latter case, Java PathFinder (JPF) [103] is used for the actual verification. An exception is [13], which currently takes a special place in the agent verification literature: it is the only publication that presents a state space reduction algorithm tailored to an agent programming language (namely AGENTSPEAK).

Previously mentioned papers can be regarded as the *first wave* of APL model checking. Characteristic to the first wave is the translation of the code of an agent (e.g. in MABLE or AGENTSPEAK) to the input language of an existing model checker (e.g. PROMELA/SPIN or JAVA/JPF), which is then used to perform the actual verification. However, this is not as straightforward as it seems. One of the issues that arises is the complexity of encoding agent concepts (such as beliefs, goals, desires, and intentions) in the input language of a non-agent-oriented verification tool (e.g. argued in [9]).

The *second wave* of APL model checking aims at resolving this issue, and is marked by the start of the MCAPL project (MCAPL is an acronym for *model checking agent programming languages*) in the summer of 2006. Drawing from previous experience and the difficulties encountered during the translation of AGENTSPEAK to PROMELA and JAVA, the MCAPL project aims at the construction of a generic JAVA infrastructure layer in which various (BDI-based) agent languages can be implemented, interpreted, and verified with a generic model checker. The JAVA layer is called the *agent infrastructure layer* (AIL) [32, 33], and provides an agent language designer with the necessary tools to implement a BDI language with customisable semantics. Moreover, all languages that are implemented in the AIL can be verified with the generic AIL model checker, called *Agent* JPF (AJPF) [67, 9], which is an extension of JPF. As such, the complex encoding of agent concepts in the input language of an existing model checker need be done only once, namely during the implementation of AJPF, after which all languages implemented in the AIL benefit from this effort. Among the languages that at current have an AIL implementation is a (subset of) GOAL [31]. We discuss this in more detail later.

Concurrent with the MCAPL project, the MAUDE term rewriting language [26] was advocated for prototyping, testing, and model checking agent programming languages [98, 97]. One of the claimed advantages of the use of MAUDE is that it provides a single framework in which the use of a wide range of formal methods is facilitated [97]. Also (quoted from [98]):

> "The language [cf. MAUDE] has been shown to be suitable both as a logical framework in which many other logics can be represented, and as a semantic framework, through which programming languages with an operational semantics can be implemented in a rigorous way."

As such, it is argued that MAUDE is very well suited for prototyping (BDI-based) agent programming languages, and verifying agent programs with MAUDE's built-in LTL model checker [37]. We refer to this model checker in the remainder as MMC (acronym for MAUDE *model checker*). Specifically, in [98], the 3APL language is (partially) implemented in MAUDE, and in [97], the 3APL derivative BUPL is implemented in MAUDE, and model checked with MMC. It is worth noting that a prototype of the AIL framework has been implemented in MAUDE as well [40]. Also, a (rudimentary) implementation of GOAL exists (yet unpublished); we treat this implementation in more detail later.

The publications belonging to the second wave of APL model checking are characterised by their aim at finding solutions to the encoding issues that arise when agent concepts are translated to the input language of traditional model checkers. The MCAPL project tries this by implementing a new JAVA framework, whereas the [98, 97] approach tries to apply MAUDE with its convenient logical and semantic characteristics for this purpose. Nevertheless, both the MCAPL project and the MAUDE approach at some point rely on an existing model checker for verification: in the MCAPL case, the AIL model checker AJPF is an extension of JPF, whereas in the MAUDE case, verification relies on MMC.

Thus, the approach taken in this thesis, although belonging to the line of research that focusses on APL model checking, is quite different from past efforts. We do not rely on an existing model checker or verification tool, but have implemented *almost everything* from scratch. That is, everything *except* the language interpreter. As such, we deviate from an idea that has lived in the agent model checking field already since [5], i.e. the late 1990s, in which it is claimed that we should "reuse with almost no variation the technology and tools developed in [traditional] model checking". Basically, with our approach, we turn this statement around: it is our believe that *we should reuse with almost no variation the technology and tools developed for agent programming languages.*

The reason for reusing existing tools is the idea that agent model checking as such can benefit from optimisations and state space reduction techniques that are already implemented in existing state-of-the-art model checkers. It is, however, not clear whether such benefits are actually gained. In fact, in [13], a state space reduction algorithm designed specifically for AGENTSPEAK reduces the state space of an AGENTSPEAK agent by roughly 20%, whereas a similar algorithm that ships

with Spin's default package (thus not tailored to AgentSpeak) yields no reduction at all. At the same time, however, encoding the agent programming language's semantics in the input language of a traditional model checker poses problems. On the one hand, as mentioned earlier, translating agent programs to an existing model checker's input language is complex, which is what the second wave of Apl model checkers try to resolve. On the other hand, however, there might also be performance issues instead of the expected gains. We identify two possible causes for this.

- One reason is that agent programming languages are typically defined on a higher level of abstraction than the language in which they need be encoded. For example, AgentSpeak is certainly more abstract than Promela and Java. Similarly, the agent programs interpreted by the Ail and model checked with Ajpf are verified as Java programs, hence also on a lower level of abstraction. As a consequence, verification occurs on a more detailed level than necessary, which may introduce a certain amount of overhead.

- Another reason is the level of efficiency at which the agent programming language's semantics can be implemented in the input language of the existing model checker. It is fair to assume that the standard interpreter of the agent programming language is optimised at least to some extent with respect to generation of states. It is, however, questionable whether an implementation of the semantics (either as a separate interpreter, or encoded in the translation of the agent) in the existing model checker's input language is just as efficient. As successor generation is a very frequently occurring operation in model checking, it is imperative that it is performed with as little resource consumption (in terms of both time and memory) as possible.

Whereas the prevailing idea is that model checking is more difficult (from an implementation point of view) than agent interpretation (hence the reuse of existing model checkers), the above suggests the converse. We *have* the technology to efficiently interpret agent programs at the appropriate level of abstraction, and, in our experience, it is easier to implement model checking algorithms from scratch such that they can benefit from these existing interpreters than to try to encode an efficient interpreter from scratch in an often too inexpressive language (e.g. Promela) for this purpose.

So far, we have mainly discussed efficiency benefits of using the standard interpreter, but it is worth noting that our approach also enables a more expressive property specification language. In this respect, we do conform to one of the proposals made in [5], namely that "when we consider the temporal evolution of an agent we treat Bdi atoms (i.e. atomic formulas expressing belief, desire, or intention) as atomic propositions". In fact, all the above referenced papers on Apl model checking define LTL-based specification languages, whose atomic propositions are Bdi atoms (i.e. mental state conditions in case of Goal). However, in all these publications, constraints are imposed on the Bdi atoms that the corresponding model checker allows. For example, in [9], Bdi atoms may not contain free variables (e.g. the Goal belief atom **bel**`(ticked(X))` is not allowed), and they may not contain inner structure (e.g. the Goal belief atom **bel**`(ticked(bananas), ticked(oranges))` is not allowed). In contrast, in our verification language, no constraints need be placed on the allowed mental state conditions, i.e. *any* Msc that the interpreter can deal with may occur in a property (this includes both of the aforementioned belief atoms). Clearly, this is beneficial.

Finally, we remark that no complex encoding of semantics or translation of agents is necessary in our approach. Because we use the *standard* interpreter, we can use the *standard* parser, hence agents written in the *standard* syntax can be model checked. Additionally, all language features and constructs that the interpreter can handle are immediately verifiable as well, although we acknowledge that some language-specific optimisations may require additional effort in this respect. For example, if a new Goal construct fundamentally changes how mental state are represented, then we might need to revise the binary mental state optimisation.

To summarise, we attribute the following benefits to our approach:

1. Improved performance in terms of resource consumption during model checking (to be shown experimentally; see below).
2. Increased expressiveness of properties.
3. Immediate and encoding-less language support.

Are there also drawbacks? Well, implementing a model checker from scratch and getting acquainted with the theory takes time. In the words of [41]: "the prevalent way of developing a software model checker for a specific programming language is a labor-intensive process often requiring man-years of effort". Note, however, that in our approach, an important part of the model checker already exists: the interpreter. Hence, we "merely" needed to implement algorithms for checking language emptiness of Büchi automata and LTL-to-Büchi translation. As these implementations at current exist, re-using our framework for model checking another language, e.g. AgentSpeak, would only require to re-implement the program component as a bridge between the exploration component and the AgentSpeak interpreter. This does not have to take man-years of effort. To illustrate, developing the code that connects the Goal interpreter to the program component, including the binary mental state optimisation, took us approximately one man-month of work.

We acknowledge that the first benefit listed above is quite a blunt statement without evidence. Therefore, in the remainder of this chapter, we present a quantitative performance comparison of three Goal model checkers (including ours) in support of this claim.

4.2 Performance Comparison: Experimental Design

As remarked earlier, subsets of Goal have been implemented in the Ail and Maude, such that we can compare the performance (in terms of resource consumption during verification runs) of our approach with current state-of-the-art alternatives.

Before turning to the experiments performed (see Sect. 4.3), however, we treat in more detail the general experimental set-up in this section. We start with a description of the other Goal model checkers, continue with the issues that arise when we want to come to a *fair comparison*, and finally treat the means with which we measure and compare resource consumption.

4.2.1 Participants

The experiments feature three *participants*: Ajpf (with the Ail as execution environment), Mmc (with Maude as execution environment), and our own approach, which we term *interpreter-based model checker* (Imc) in the remainder, and whose execution environment is the standard Goal Java/Pl interpreter. While in Chap. 3, we have discussed Imc in detail, Ajpf and Mmc have only been briefly mentioned in the previous section; here, we provide a more detailed account.

Ail and Ajpf

As outlined in the previous section, the Ail is a Java framework in which Bdi-based agent languages can be implemented. We treat here the Ail's Goal implementation, in the following referred to as "Ail Goal".

The Ail Goal interpreter was first presented in [31], and offers different functionality than the standard Goal interpreter. The most notable difference is the expressiveness of the language of mental state conditions: the mental atoms that are allowed to occur in an Ail Goal agent may not contain inner structure (e.g. the Goal belief atom **bel**`(ticked(bananas), ticked(oranges))` is not allowed), whereas in the standard Goal interpreter, *any* Pl query may be a **bel** operator's argument (including those that contain built-in Pl predicates). Also, the semantics of the **goal** operator *seems* different:

Standard interpreter:	$\langle K, \Sigma, \Gamma\rangle \models_\Psi \mathbf{goal}(\chi)$	iff	there exists a $\gamma \in \Gamma$ such that $K \cup \Gamma \models \chi$
Ail interpreter:	$\langle K, \Sigma, \Gamma\rangle \models_\Psi \mathbf{goal}(\chi)$	iff	$\chi \in \Gamma$

However, a goal in AIL GOAL is a single fact, and a knowledge base does not exist. Thus, in AIL GOAL, the goal base may be regarded as a set of facts, in essence similar to the belief base, such that the semantics of the **goal** operator is not actually different: the restricted type of goals that are featured in AIL GOAL just enable a simpler definition of its semantics. In contrast to the part of GOAL that we consider in this thesis, AIL GOAL does allow for (heterogeneous) multi-agent systems and the incorporation of environments. Finally, although not in the AIL documentation, during design of our experiments we found that the precondition of user-defined actions must be a single literal, and may not be a conjunction of these. Also, we found that achieved goals are not removed *at the end* of a reasoning cycle (after an action has been performed), but rather *at the beginning* of the next. This indeed leads to slightly different semantics (i.e. transition systems); we discuss the implications of this to our experiments in more detail later.

Like our verification logic, the property specification language used to express properties about AIL GOAL agents is based on LTL, although the X operator is not featured [9]. The atomic propositions are, similar to ours, instantiated as a set of mental state conditions, but the same restrictions on MSCs as the ones imposed on the MSCs that may occur in an AIL GOAL agent apply: mental atoms are not allowed to have inner structure. Additionally, mental atoms that occur in properties must be ground, i.e. they may not contain free variables, e.g. the belief atom **bel**`(ticked(X))` is not allowed [9]. Note that our property specification language allows the entire language of mental state conditions, including mental atoms with inner structure and free variables.

The AIL's model checker, AJPF, is an extension of JPF. Essentially, JPF is a JAVA virtual machine implemented in JAVA (thus running on *another* virtual machine) that explores all execution paths of a JAVA program instead of only a single path as a "normal" virtual machine does [103]. JPF has undergone quite the evolution over the years. It started as an LTL model checking project for JAVA programs, in which these programs were translated to PROMELA and verified with SPIN [49]. Indeed, this is quite similar to the MABLE and AGENTSPEAK efforts, which constituted the first wave of APL model checking. One of the reasons for the JAVA-to-PROMELA approach to grow out of favour was the complexity of the translation of JAVA language constructs that are not featured natively by PROMELA, e.g. floating point numbers [103] (again, note the parallels with the first wave of APL model checking). To overcome these problems, a custom model checker for JAVA was built (in JAVA), natively supporting deadlock detection and assertion/exception checking, but *not* generic LTL property verification. That is, the current JPF does not support an LTL-based specification language (in industry, JPF is mainly used for debugging rather than for verification [9]).

Because JPF is open source, it can be extended easily. The first step in the construction of AJPF was to add LTL property verification to it. To this end, the GPVW algorithm, a predecessor of the LTL2AUT algorithm that we use, has been implemented for LTL-to-Büchi translation (the use of GPVW is not remarked in [9]; we found this by inspection of AJPF's code). Although AJPF is claimed to be an explicit-state automata theoretic model checker, its implementation is slightly opaque in this respect. From the sparse information in [9], the property automaton resulting from GPVW is run, as a JAVA program, alongside the program automaton, which is also a JAVA program (namely the AIL interpreter), resulting in the product automaton, which is hence a big JAVA program. Subsequently, JPF is used to explore all execution paths of this program, and by means of *listeners* (with which the progress of JPF can be monitored) satisfaction of the LTL property is checked.

Because JPF is not the fastest model checker around ("SPIN is at least an order of magnitude faster than JPF" [103]), special attention has been given to optimising it for verification of agent systems during AJPF's (ongoing) development. In particular, because agent programs are verified with AJPF as if they are "normal" yet complex JAVA programs, a state in the state space of the product automaton is not a straightforward pair of a program state and a product state (as in our approach), but rather a collection of information about JAVA threads and variables, i.e. a JPF state [69]. Indeed, in principle, after the execution of every JAVA byte code instruction JPF reaches a new state (JPF already comes with some optimisation techniques to reduce this though). The recording of all these states takes time, and the storage of all these states requires memory,

even though in agent verification, most of these Jpf states do not really matter. In fact, one of the perils of agent-oriented programming is the abstraction of all these low level details by defining the operational semantics on a higher level.

To record only the states that *do* matter, i.e. the Jpf states that are reached after the execution of an entire reasoning cycle of an agent, Ajpf executes as much of the Java code as possible *atomically*: blocks of Java code that are marked atomic are executed by Jpf without intermediate recording and storage of states. In [9], it is shown that very significant improvements in performance are gained. Still, there is not an exact one-to-one correspondence between the states that Jpf records with this optimisation and the product states in our approach. One of the reason is that Ajpf *always* assumes an environment, albeit empty. We illustrate this further during the treatise on the experiments.

Maude and Mmc

The Maude implementation of Goal (yet unpublished), in the following referred to as "Maude Goal", was originally developed to see if Prolog could be implemented in Maude. In the end, however, this project yielded a rudimentary interpreter for Goal, which we extended such that it would support sufficient functionality to participate in the experiments that follow. That is, the first version of the Maude Goal interpreter only facilitated verification of Goal agents consisting solely of user-defined actions. We added support for **adopt** and **drop** actions such that at current, Maude Goal is not much unlike the single-agent non-environment subset of Goal that we consider in this thesis: in contrast to Ail Goal, mental atoms in Msc-s may have inner structure and free variables (both in the agent code as well as in the property specification), action preconditions are (in general) conjunctions (rather than single facts), and the same holds for the representation of goals.

Let us have a closer at the differences. First, only a very small subset of Pl built-in operators and predicates can be used in Maude Goal (i.e. conjunction and negation). Another difference, though not directly related to functionality of the language, is that the syntax of Maude Goal is plain Maude. That is, whereas Ail Goal defines an Ail-style syntax to define Ail Goal agents in, agents written in Maude Goal need be specified in Maude, yielding programs that are not straightforwardly readable: familiarity with Maude is required (an automatic Goal-to-Maude translation tool in not available). Also, to define the knowledge base of an agent, direct calls to the Maude implementation of Pl are required. Hence, it is debatable whether Maude Goal natively supports knowledge bases, or merely gives the opportunity to "hack" them into the agent. Last, but certainly not least, there is a quite significant difference in semantics with respect to removal of achieved goals. We already saw that Ail Goal also differs from standard Goal in this respect, as it removes achieved goals at the beginning of the agent's next reasoning cycle rather than at the end of the current. Maude Goal applies yet another approach; we treat this in more detail later.

As before, the property specification language is LTL whose propositions are mental state conditions. Verification of Maude Goal programs occurs with Maude's built-in LTL model checker (i.e. Mmc). This model checker was presented in [37], and is, like ours, of the on-the-fly Ndfs explicit-state automata-theoretic kind. The used algorithms are, however, different. With respect to LTL-to-Büchi translation, Mmc applies the Ltl2Ba algorithm of [43]. This is one of the algorithms that succeeds Ltl2Aut (which we use) chronologically, but whose possible superiority has not yet been established (according to [100] on which our choice is based). In addition, Mmc applies a series of LTL equivalences (namely those presented in [39, 89]) to simplify the property before it is translated to a Büchi automaton. We have not (yet) implemented such optimisations. The Ndfs algorithm implemented in Mmc is based on [60], which in turn is based on [27], the first publication on Ndfs algorithms. In [100], better performance is ascribed to the algorithm that we use.

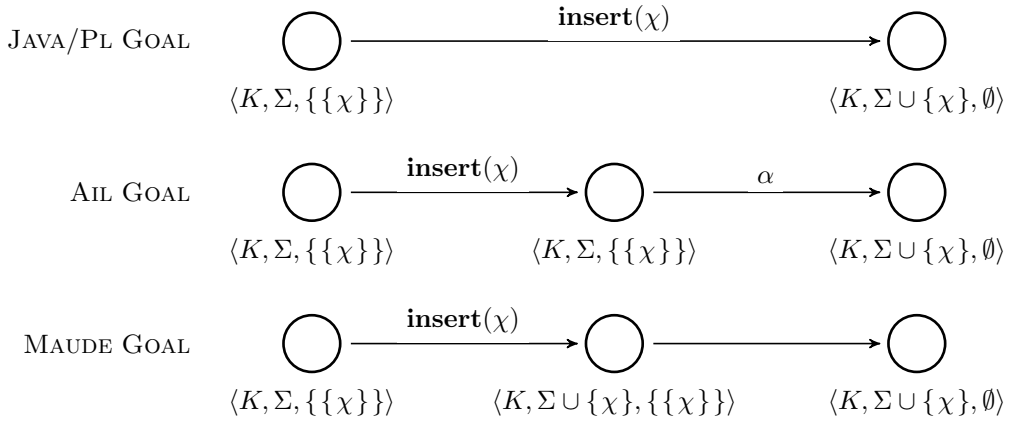


Figure 4.1: Differences in goal removal between Imc, Ajpf and Mmc.

Goal removal

Before summarising the (dis)similarities of the 3 model checkers, we first discuss the different goal removal strategies in more detail. Recall that the standard interpreter removes achieved goals immediately after the action is performed. In contrast, in Ail Goal, an achieved goal is removed at the beginning of the next reasoning cycle. The approach taken by Maude Goal is the introduction of an *intermediate mental state* that is not generated by the other two interpreters. We illustrate the different removal strategies with Fig. 4.1, and the following explanation.

Suppose an agent has the single goal to bring about χ, and an arbitrary knowledge base K and belief base Σ such that $K \cup \Sigma \not\models \chi$. After the addition of χ to the belief base (e.g. by means of an **insert**(χ) action as in Fig. 4.1), the standard Java/Pl Goal interpreter will immediately update the goal base, resulting in the mental state in which **bel**(χ) is true and **goal**(χ) is false.

The Ail Goal interpreter operates roughly the same, except that the mental state (including the goal base) is updated at the beginning of the next reasoning cycle, in which some action α is performed later. Because carrying out these updates are the first thing that happens, the evaluation of the precondition of α (and the Mscs in action rules) occurs in the same mental state as the mental state in which these Mscs would be evaluated when the standard interpreter is used. The changes that α brings about are subsequently reflected in the mental state after performance of the execution following α, et cetera. As a consequence, the computations generated by the Ail Goal interpreter are comparable to the computations generated by the standard Goal interpreter.

Finally, the Maude Goal interpreter introduces an intermediate mental state that is not generated by the other two interpreters: in this mental state, both **bel**(χ) and **goal**(χ) are true. From this intermediate state, an anonymous action removing the achieved goals is always performed first, if applicable. That is, the anonymous action can be thought of as having priority over all other actions, such that actions that would be enabled in the intermediate mental state are never performed: the mental state is updated before these actions are even considered. Thus, in practice, a Goal agent interpreted by the Maude Goal interpreter does not behave differently; it only has these intermediate mental states. Note that, of course, both the standard and the Ail Goal interpreters at some point during the reasoning cycle have such an intermediate mental state as well. However, in these interpreters, this state is not visible to the outside, i.e. not a state in the transition system. The Ail Goal interpreter "hides" this mental state quite explicitly by marking the Java code that implements a reasoning cycle as atomic.

The above differences have implications for the properties that we consider in our experiments. We treat this in the next subsection.

Summary

In this subsection, we have treated in some detail the characteristics of Ail Goal, Mmc Goal, and their model checkers. The differences and similarities are summarised in Tables 4.1 and 4.2.

4.2.2 Experimental Agents

The experiments are designed to enable a fair comparison. The most important condition for this is that semantically equivalent Goal programs are provided as input to the three different model checkers. Because the Ail Goal interpreter, the Maude Goal interpreter, and the standard Java/Pl Goal interpreter do not feature the same functionality, it turned out to be rather complex to design experiments. As a result, the experimental agents are simple and may appear artificial in some respects; unfortunately, the differences could not be overcome otherwise.

A second condition for a fair comparison is that the properties that we verify are insensitive to the different semantics with respect to removal of achieved goals. That is, the truth or falsehood of properties under investigation should not depend on the presence or absence of the intermediate mental state in Maude Goal, or the fact that computations generated by the Ail Goal interpreter are "one step behind". Therefore, we only check properties of the form $\mathsf{Fbel}(\chi)$, and include an analysis of the state space (of the product automaton) as explored by the model checkers for the sake of clarification.

Apart from these considerations concerning semantics, there has been another reason for keeping the experimental agents simple: if performance of the model checkers is already poor with simple agents (which is, as we will see, for some participants the case), it will only be worse when the scenarios are more complex. This has motivated us to only consider *deterministic agents*, i.e. agents that have a single computation, in our experiments. This decision was made already during the initial phase of experimental design, in which we found that even simple non-deterministic agents are beyond the capabilities of Mmc, and to a lesser extent Ajpf.

	Java/Pl interpreter	Ail interpreter	Maude interpreter
System type	Single-agent*	Multi-agent	Single-agent
Environment	No*	Always	No
Mental atoms:			
• in agents	Any Pl query	No inner structure	Restricted Pl query
• in LTL	Idem	Idem + no free variables	Idem
Precondition	Conjunction of literals	Single fact	Conjunction of literals
Goals	Conjunction of facts	Single fact	Conjunction of facts
Goal removal	Immediately	In next reasoning cycle	With intermediate state

* The standard Goal interpreter in principle does support environments and multi-agent systems, but because we have not discussed this functionality as the scope of this thesis is single-agent systems without environments, we assume this functionality absent here.

Table 4.1: Differences between the standard Java/Pl Goal interpreter, the Ail Goal interpreter, and the Maude Goal interpreter

	Imc	Ajpf	Mmc
State generation	Java/Pl interpreter	Ail interpreter	Maude interpreter
LTL-to-Büchi	Ltl2Aut	Gpvw	Ltl2Ba with [39, 89]
Exploration	Ndfs of [92]	Jpf-based	Ndfs of [60, 27]

Table 4.2: Differences between Imc, Ajpf and Mmc.

Another benefit of considering deterministic agents, is that it enables us to compare resource consumption during verification with resource consumption during normal execution of an agent in a straightforward fashion. Such a comparison cannot be done with non-deterministic agents, while it does give insights in the workings of a model checker and the overhead it imposes. For example, if normal execution is already quite slow, then it is to be expected that verification will also be slow. Conversely, if there is a large difference between verification and execution times, then this might be an indication that the model checker is operating at suboptimal efficiency.

4.2.3 Measuring and Comparing

The last consideration concerning experimental design is how results can be (i) obtained, i.e. measured, and (ii) analysed, i.e. compared.

Measuring results

As we want to compare performance, the obvious choice is to measure resource consumption of the model checkers along two dimensions, called *dependent variables* in statistics (e.g. [84]): verification time and memory consumption. Verification time is straightforwardly measured, in milliseconds, by recording the start and finish times of the model checkers, and computing the difference between them. More elaborate are the measurements on memory consumption.

Both IMC and AJPF are implemented in JAVA, and their memory consumption is measured by inspecting the size of the heap space. Specifically, the maximum size of the heap during execution is reported as the memory consumption. This way of measuring was observed in and adopted from JPF. Unfortunately, by reporting memory consumption this way, an *upper bound* on the consumption is obtained rather than the minimally required amount. This is due to the garbage collection algorithm of the JAVA virtual machine, combined with the supplied maximum heap size (by means of the `-Xmx` argument); if not explicitly instructed to do so, garbage collection is only carried out if insufficient memory is available for allocation of a new object. Due to this, we do not regard the measurements on memory consumption for IMC and AJPF as completely accurate, but rather as a rough indication. For MMC, the measurements were recorded by inspecting the total amount of memory reserved for the MAUDE process by the operating system (in our case Linux) during model checking. In a public e-mail exchange (see `http://lists.cs.uiuc.edu/pipermail/maude-help/2010-March.txt`), it is acknowledge by Steven Eker, one of MAUDE's designers and writer of [37], that for measuring MAUDE's memory consumption: "`top` is probably as good as anything" (`top` is a Linux utility with which processor activity and memory consumption of running processes can be monitored).

Analysing results

Measuring alone is insufficient to draw conclusions. That is, we are particularly interested in the *scalability* of the model checkers, especially since the agents we consider in our experiments are fairly simple. However, "scalability" is a qualitative term, which we somehow must quantify to make meaningful statements. Although improving scalability has been an important factor in research on software verification [25], to the best of our knowledge, no quantitative standards or metrics regarding scalability have been proposed for model checking, although the following is remarked in [19] (in the context of hardware model checking):

> "Improving scalability requires more than just a constant factor speed up; it requires a drastic reduction in the rate that verification time increases as a function of increasing circuit size (for example, exponential growth reduced to quadratic growth)."

To clarify the notion of scalability (at least in this thesis), we will say that a model checker is *scalable* or *scales well* in the experimental conditions if the relation between these conditions and resource consumption can be described by one of the following functions:

$$\begin{array}{lcl} \text{Logarithmic} & : & y = b \cdot \log(x) + a \\ \text{Polynomial } (d < 1) & : & y = b \cdot x^d \\ \text{Linear} & : & y = b \cdot x + a \end{array}$$

Conversely, a model checker is said to be *not scalable* or *scale poor* in the experimental conditions if the relation between these conditions and resource consumption can be described by one of the following functions:

$$\begin{array}{lcl} \text{Polynomial } (d > 1) & : & y = b \cdot x^d \\ \text{Exponential} & : & y = b \cdot r^x \end{array}$$

In these functions, x and y represents, respectively, the conditions and the resource consumption, a is called the *intercept*, b is called the *coefficient*, d is called the *degree* of the polynomial, and r is called the *radix*.

The reason for regarding polynomial relations in degree $d > 1$ as not scalable is that x is, in general, very large for real-world model checking problems. In such cases, a quadratic relation versus a cubic relation can already make the difference between tractability and intractability of a problem. Note that our definition differs from complexity theory, in which all problems that can be solved in polynomial time are thought of as tractable (e.g. [71]). To determine the type of relation between experimental conditions and resource consumption, we will apply *regression analysis* on the measurements (e.g. [84]). That is, we will fit the functions given above to the measurements using least-squares regression, and assess the goodness of the fits by comparing their R^2 values. The fitted function that yields the highest (i.e. closest to 1) R^2 is deemed the relation between conditions and consumption.

The experimental conditions that we will vary, in statistics called *independent variables*, are the size of the belief base and the size of the state space. The size of the state space is defined as the total number of mental states that can be encountered on all computations of a Goal agent, i.e. the number of states in its transition system. The size of the belief base is defined as the number of facts it contains. To study the different effects of both independent variables, the experiments are organised as follows. In the first and second experiment, the size of the belief base is varied, while the size of the state space is kept constant. In the third experiment, the size of the belief base is kept constant, while the size of the state space is varied. Finally, in the fourth experiment, both the size of the belief base and state space are varied.

4.3 Experiments and Results

We now present and analyse the experiments carried out, as well as their results. All experiments are performed on an Intel Core2Duo T6670 (2.2 GHz) machine, running Ubuntu 9.10. The heap size of the Java virtual machine has been set to 2500 MB (i.e. `-Xmx2500M`) in all experiments that involve Java execution. The raw data from which figures in this section are derived can be found in Appx. B.

4.3.1 Experiment 1: Blocks World

In this first experiment, we investigate the scalability of the model checkers in the size of the belief base; the size of the state space is kept constant. The experiment involves a blocks world (e.g. [88]) agent with $n \in \{10, 20, 30, 45, 60, 80, 100, 200\}$ blocks; the blocks are labelled `aa`, `ab`, `ac`, et cetera. Four of the n blocks are initially stacked on each other, while the remaining $n - 4$ blocks are on the table. For all n, the stacked blocks are `aa`, `ab`, `ac`, and `ad`: `aa` is on the table, `ab` is on `aa`, `ac` is on `ab`, and `ad` is on `ac`. In the target configuration, all the blocks are on the table. Fig. 4.2 shows the code of the agent, in the remainder called `blocksAgent`, for $n = 5$, and Fig. 4.3 depicts the corresponding source and target configurations.

The belief base of `blocksAgent` comprises all information about the current configuration of the blocks, whereas the goal base contains a single goal: having `ab` on the table. That is, `blocksAgent`

```
main: blocksAgent{
  beliefs{
      block(aa). block(ab). block(ac). block(ad). block(ae).
      on(aa,table). on(ab,aa). on(ac,ab). on(ad,ac). clear(ad).
      on(ae,table). clear(ae).
  }
  goals{
    on(ab,table).
  }
  program{
    if goal(on(ab,table)), bel(on(X,Y)), bel(clear(X))
    then moveXfromYtoTable(X,Y).
  }
  actionspec{
    moveXfromYtoTable(X,Y){
      pre{ block(X) }
      post{ not(on(X,Y)), on(X,table), clear(Y) }
    }
  }
}
```

Figure 4.2: Blocks world agent for $n = 5$, called `blocksAgent`.

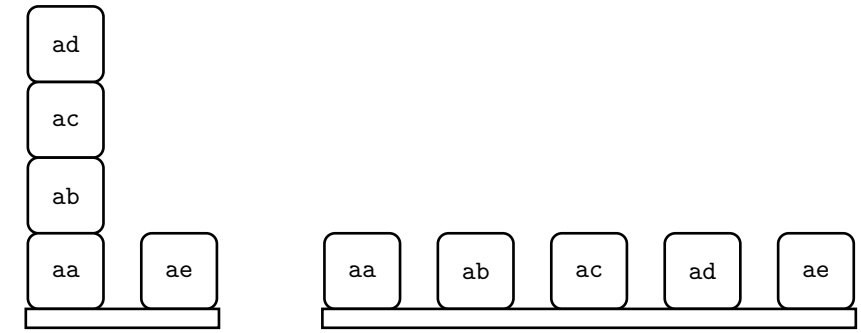


Figure 4.3: Source and target configurations for $n = 5$.

has no goals about the position of the other blocks. Though defining its goal of bringing about the target configuration this way is a bit artificial, it is necessary because AJPF does not permit conjunctions of facts as goals.

The target configuration may only be modelled as outlined above if the following conditions are met:

- Blocks are only moved from some other block to the table.
- Block `ab` is the last block to be moved to the table.

The second condition is met by the way the blocks are stacked on each other: `ab` is always the last block to be moved to the table, given that blocks cannot be moved as long as other blocks are on top of them. To satisfy the first condition, `blocksAgent` is equipped with a single action rule: whenever it still has its initial goal, and a block `X` is clear (i.e. there is no block on top of it), while positioned on some block `Y`, `blocksAgent` should move `X` to the table. Thus, only blocks that are on another block are moved, and always to the table. The postcondition of such a `moveXfromYtoTable` action is that `blocksAgent` believes that `X` is on the table rather than on `Y`, and that `Y` is clear. The precondition should enforce that all blocks to be moved are in fact blocks (and not the table).

The property φ under investigation is whether `blocksAgent` eventually believes to have brought about the target configuration. Formally: $\varphi = \mathsf{F}\,\textbf{bel}(\texttt{on(ab,table)})$ such that its negation in NNF is $\neg\varphi = \mathsf{G}\neg\textbf{bel}(\texttt{on(ab,table)})$. For all n, the structure of the transition system of `blocksAgent`

is identical, hence the state space is of equal size. In contrast, the belief base in each mental state grows as n increases: it becomes filled with *redundant* blocks. A block is said to be redundant if removing the block does not affect the behaviour of `blocksAgent`. Thus, by varying the value of n, we control the independent variable (size of the belief base), enabling an investigation of its effect on scalability.

State space

Before presenting the results of this experiment, we first have a look at what the state spaces of the product automaton of the different model checkers look like.

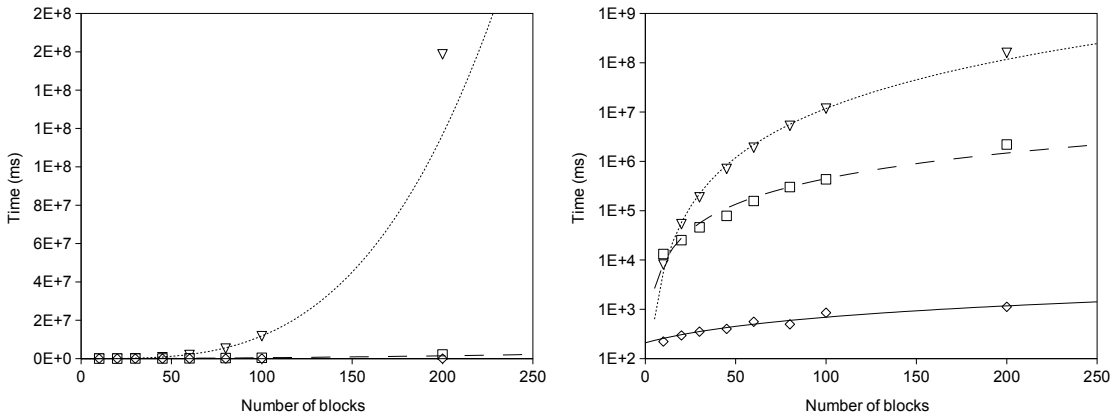
Imc Imc reports, for all n, that it has stored five states during the search, of which one is the initial source state $\langle \imath_P, \imath_{\neg\varphi} \rangle$. Let $\psi = \mathbf{bel}(\texttt{on(ab,table)})$, and let $q_{\neg\varphi} = \{\neg\psi, \mathsf{X}\mathsf{G}\neg\psi\}$. The other four product states $q^i_\times = \langle q^i_P, q_{\neg\varphi} \rangle$ in which $(1 \leq i \leq 4)$ look as follows:

$$
\begin{array}{lcl}
q^1_P & = & \langle \emptyset, \{\texttt{on(ad,ac)}, \texttt{on(ac,ab)}, \texttt{on(ab,aa)}, \ldots\}, \{\{\texttt{on(ab,table)}\}\} \rangle \\
q^2_P & = & \langle \emptyset, \{\texttt{on(ac,ab)}, \texttt{on(ab,aa)}, \ldots\}, \{\{\texttt{on(ab,table)}\}\} \rangle \\
q^3_P & = & \langle \emptyset, \{\texttt{on(ab,aa)}, \ldots\}, \{\{\texttt{on(ab,table)}\}\} \rangle \\
q^4_P & = & \langle \emptyset, \{\ldots\}, \emptyset \rangle
\end{array}
$$

In the above, "..." represents all beliefs about blocks that are already on the table.

Ajpf Ajpf reports, for all n, that it has stored 18 states. Due to the opaqueness of the way the product automaton is constructed with Ajpf, it is difficult to characterise each state as we did for Imc. Therefore, we proceed with a more procedural exposition. In the analysis that follows, we denote states by $state_i$, where i is the index of the state as given by Ajpf.

1. The first state, $state_0$, is obtained after initialisation of the system.
2. In the three states that follow (i.e. $[state_1 \rightarrow state_2 \rightarrow state_3]$), `blocksAgent` starts reasoning, after which it performs an action (moving `ad` to the table). Thus, `blocksAgent`'s first reasoning cycle is spread over three states (despite the claim of [9] that each reasoning cycle is executed atomically). At the end of $state_3$, `blocksAgent` still believes that `ad` is on `ac`; the update to the belief base is made at the beginning of the next reasoning cycle (this means that $state_5$ is the first state in which `blocksAgent` believes `ad` to be on the table).
3. Between $state_3$ and $state_4$, the (empty) environment performs a reasoning step.
4. The sequences $[state_4 \rightarrow state_5 \rightarrow state_6]$ and $[state_6 \rightarrow state_7 \rightarrow state_8]$ are similar to Items 2,3. That is, `ac` and `ab` are moved to the table, respectively, and between the reasoning cycles of `blocksAgent`, the environment is given the opportunity to reason. In contrast to the sequence $[state_0 \rightarrow (state_1 \rightarrow state_2 \rightarrow state_3) \rightarrow state_4]$, the reasoning cycles of `blocksAgent` are no longer spread over three states, but over only a single (in compliance with [9].
5. In the reasoning cycle that starts after $state_8$, Ajpf first updates the belief base (`blocksAgent` then believes that all the blocks are on the table), and finds out that $\neg\varphi$ cannot be satisfied any more from that point on. As a consequence, Ajpf backtracks to $state_2$. States on the path from $state_2$ to $state_8$ are all backtracked, because neither `blocksAgent` nor the environment have successor states other than the one that already was explored.
6. The first time $state_2$ was encountered by Ajpf, it continued with $state_3$ (see Item 2). At this point during the model checking run, Ajpf pursues another possibility, namely by first letting the environment reason (rather than finishing the reasoning cycle of `blocksAgent` by proceeding with $state_3$). This results in the sequence $[state_2 \rightarrow state_9 \rightarrow state_5]$: between $state_2$ and $state_9$, the environment takes an



(a) Linear scale on y-axis

(b) Logarithmic scale on y-axis

Figure 4.4: Verification times of IMC (continuous line, measurements as ◇), AJPF (dashed line, measurements as □), and MMC (dotted line, measurements as ▽) in Experiment 1. Plotted lines are best-fit regression lines.

(empty) reasoning step, and between $state_9$ and $state_5$, `blocksAgent` moves `ac` to the table. Since $state_5$ has already been encountered, it is unnecessary to continue the search from this state. Hence, AJPF backtracks again, this time to $state_1$.

7. The remaining eight states are all due to similar behaviour as outlined in the previous item: AJPF considers various interleavings of whether `blocksAgent` or the environment reasons first after reaching $state_1$. This results in the exploration of five new states. Eventually, AJPF backtracks to $state_0$ and repeats this behaviour again, yielding the remaining three states.

MMC MMC reports, for all n, that it has examined four states. The first three of these states are identical to the states $q^1_\times$, $q^2_\times$, and $q^3_\times$ that IMC encounters. The last state is, however, different: it corresponds to the intermediate mental state that occurs when a goal is achieved. That is, in the fourth state, MMC had `on(ab,table)` both as belief and as goal. The anonymous action that would follow and remove this goal is, however, not performed, because the state in which `on(ab,table)` is part of the belief base and the goal base already proves $\neg\varphi$ unsatisfiable on the current path. Hence, it was unnecessary for MMC to continue the traversal.

Verification time

The verification times are displayed in Fig. 4.4.

For IMC, the relation between number of blocks and verification time is polynomial in degree 0.55 ($R^2 = 0.9378$) or linear ($R^2 = 0.9329$); we assume the linear relation (i.e. the worst case). For AJPF, the relation between the number of blocks and verification time is polynomial in degree 1.7 ($R^2 = 0.9706$), and the second-best fit is an exponential function ($R^2 = 0.9132$); due to the large difference in R^2, we assume a polynomial relation. For MMC, similar results are obtained, i.e. the best-fit function is polynomial in degree 3.3 with a nearly perfect fit ($R^2 = 0.9958$), whereas the second-best fit is obtained when fitting an exponential function ($R^2 = 0.8594$); obviously, the polynomial relation is assumed. For $n = 200$, the absolute difference in verification time is the largest: IMC took 1 second, AJPF took 40 minutes, while MMC took as much as 44 hours.

These results suggest that IMC scales well to larger belief bases, in contrast to AJPF and MMC. Though the latter two both seem to grow polynomially in the size of the belief base, the degrees of the fitted functions tell that the increase in verification time of MMC is more than cubic, whereas

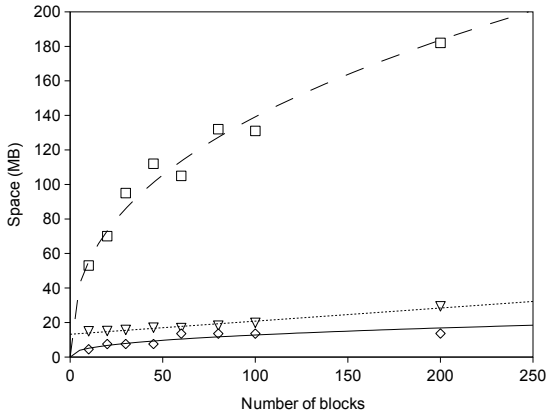


Figure 4.5: Memory consumption of Imc (continuous line, measurements as ◇), Ajpf (dashed line, measurements as □), and Mmc (dotted line, measurements as ▽) in Experiment 1. Plotted lines are best-fit regression lines.

Ajpf remains just under quadratic. This difference can also be observed in Fig. 4.4, and supports our decision to classify polynomial relations in degree $d > 1$ as not scalable.

Memory consumption

The memory consumption is displayed in Figure 4.5.

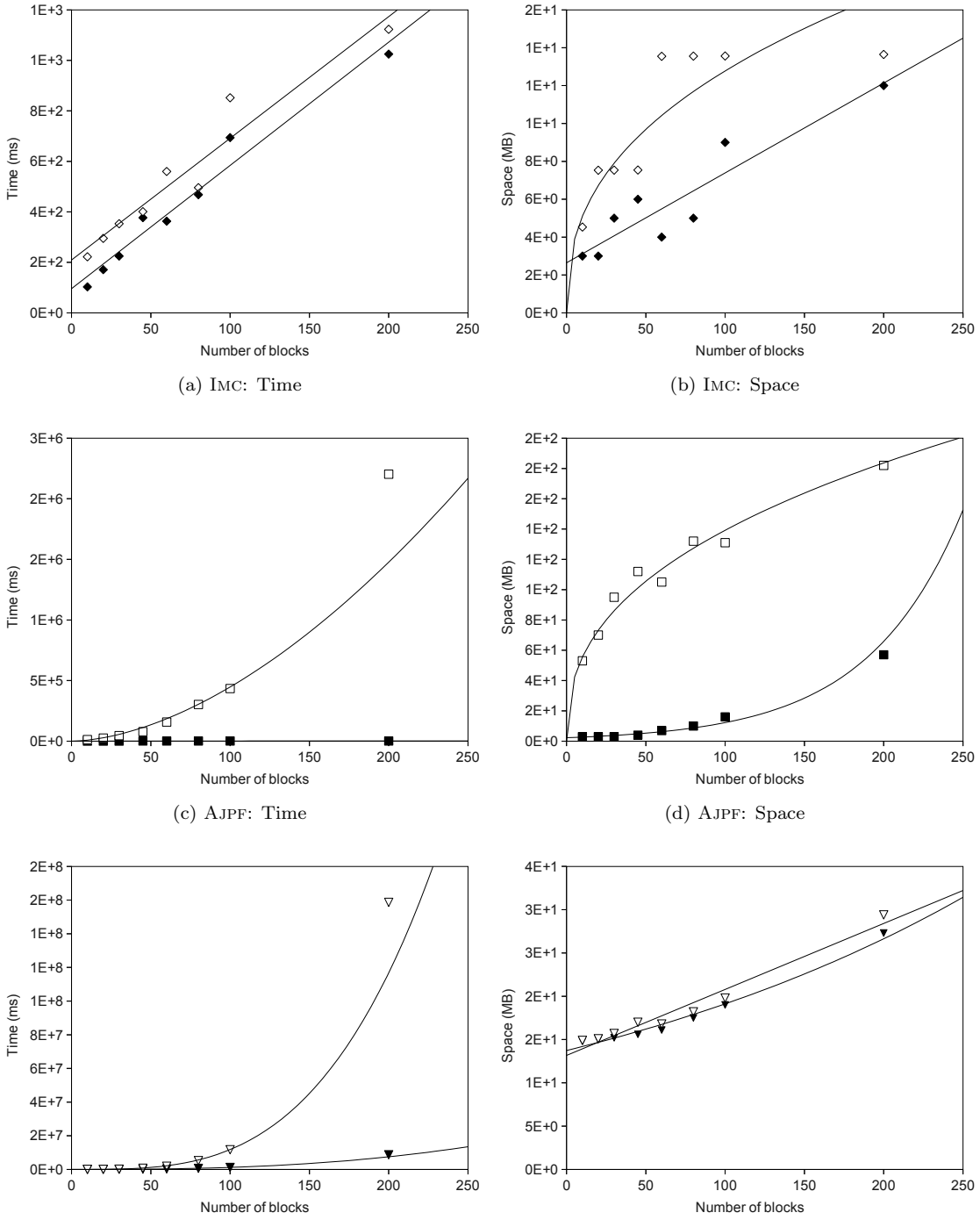
For both Imc and Ajpf, the relation between the number of blocks and memory consumption is polynomial in degree 0.40 ($R^2 = 0.8266$ and $R^2 = 0.97$, respectively). The second-best fit is, for both model checkers, obtained when fitting a logarithmic function ($R^2 = 0.8062$ and $R^2 = 0.9488$, respectively). Although the differences are slim, we assume the best-fit functions, i.e. polynomial (which are less optimal than logarithmic relations), for both model checkers. For Mmc, the best-fit function is exponential ($R^2 = 0.9878$), whereas the second-best fit is linear ($R^2 = 0.9664$). In this case, we favour the second-best fit, because the radix of the fitted exponential function, which yielded a higher R^2 value, is very close to 1, making it behave almost like a constant.

Though these values suggest that all the three model checkers scale well to larger belief bases with regard to memory consumption, it is clear from Fig. 4.5 that this notion of scalability is not the only condition for *good performance*. Indeed, although Ajpf scales well in terms of our own requirements, it goes a bit far to say that it also *performs* well, especially when compared to the memory requirements of Imc and Mmc. The reason that Ajpf is significantly more demanding than the other two environments is most likely that Jpf requires the storage of a lot more information (i.e. about Java threads and variables) than is needed for *agent verification*, hence not stored by Imc and Mmc. Also, Ajpf incorporates the (albeit empty) environment, and explores interleavings of the first few states that are not present in the other two model checkers. One might notice that although the degree of the fitted polynomial functions for Imc and Ajpf are the same, the memory requirements of the former are much lower. The reason for this is that the coefficient b for Imc is roughly 2, whereas for Ajpf it is roughly 22. The absolute difference in memory consumption is the largest for $n = 200$: Imc consumed 14 MB, Ajpf as much as 182 MB, and Mmc approximately 29 MB.

Execution

A comparison of resource consumption during model checking and normal execution is given in Fig. 4.10.

With regard to runtime, both Ajpf and Mmc introduce substantial overhead, whereas the difference is negligible for Imc (smaller than 200 ms). With regard to memory consumption,



(a) IMC: Time

(b) IMC: Space

(c) AJPF: Time

(d) AJPF: Space

(e) MMC: Time

(f) MMC: Space

Figure 4.6: Resource consumption during verification and execution of IMC (measurements as open and closed $\diamond$ for verification and execution, respectively), AJPF (measurements as open and closed $\square$ for verification and execution, respectively), and MMC (measurements as open and closed $\triangledown$ for verification and execution, respectively) in Experiment 1. Plotted lines are best-fit regression lines.

normal execution is less demanding than verification for all model checkers, although the overhead of Mmc is a lot less substantial than Imc's and Ajpf's overhead. The presence of overhead with respect to memory consumption is not very surprising, as model checking requires maintaining additional data structures during the nested depth-first search.

Intermediate conclusion

In this experiment, the scalability of the model checkers with regard to the size of the belief base was investigated. Given the measurements and fitted functions, it seems that Imc scales and performs best under the imposed conditions. Regarding the other two model checkers, Ajpf performs the worst with regard to memory consumption, whereas Mmc performs the worst with regard to verification time. Table 4.3 gives a summary.

However, we note that Ajpf explored interleavings of the execution of `blocksAgent` and the (empty) environment that were not investigated by the other two model checkers. That is, the state space that the Ail Goal interpreter explored in this experiment is a branching structure rather than a single path, despite the fact that `blocksAgent` is deterministic. Thus, even though the agents under investigation were identical, one can argue that the performance of Ajpf is not really comparable to the performance of the other model checkers due to the different nature of the state space that it needed to explore. It would, therefore, be interesting to see an experiment in which all model checkers really traverse a single path only. This is what will do in the next experiment.

4.3.2 Experiment 2: Buggy Blocks World

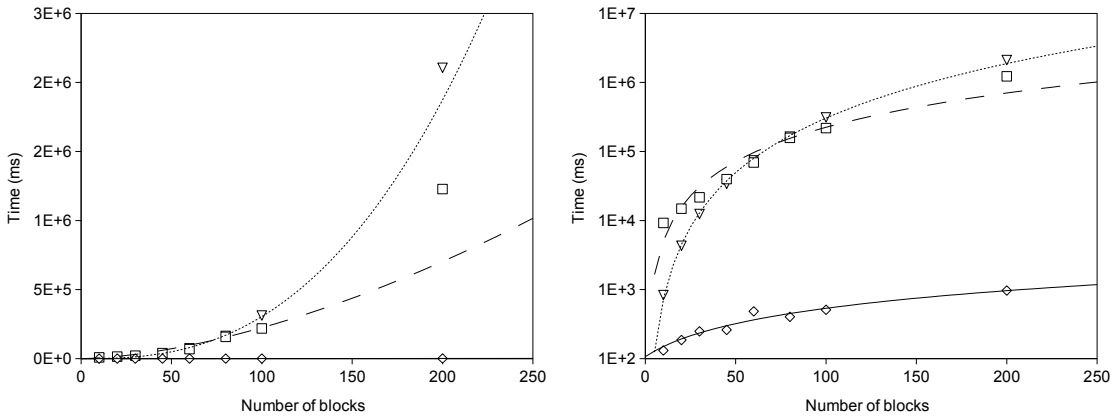
This experiment is a variation on Experiment 1, in which we enforce that all model checkers traverse a single path only (see the intermediate conclusion above for the reason why). Thus, we still vary the size of the belief base while keeping the size of the state space constant. To achieve this, rather than verifying a correct version of `blocksAgent`, a *buggy* version is under investigation: instead of the initial goal of having `ab` on the table, it has the goal of having `ac` on the table. That is, in the initial goal base (see Fig. 4.2), we substitute `on(ab,table)` for `on(ac,table)`, and we update the action rule in the **program** section correspondingly. Consequently, as soon as `ac` has been brought to the table, the action rule is no longer applicable, `blocksAgent` will perform **skip** indefinitely, and as a result never bring about the target configuration. That is, the negated property $\neg\varphi = \mathsf{G}\neg\mathbf{bel}(\texttt{on(ab,table)})$ is satisfied such that the property $\varphi = \mathsf{F}\mathbf{bel}(\texttt{on(ab,table)})$ is violated. In the remainder, we refer to the buggy version of `blocksAgent` as `buggyBlocksAgent`.

State space

Imc Imc reports, for all n, that it has stored four states, and provides a counterexample in which they all occur. The explored states are identical to the first four states that Imc explored during Experiment 1, i.e. the product source state, followed by the product state corresponding to the initial mental state, followed by the product state corresponding to the mental state after performing `moveXfromYtoTable(ad,ac)`, followed by the product state corresponding to the mental state after performing `moveXfromYtoTable(ac,ab)`.

Model checker	Verification Time Relation	Scalable	Memory Consumption Relation	Scalable
Imc	Linear	Yes	Polynomial ($d = 0.40$)	Yes
Ajpf	Polynomial ($d = 1.7$)	No	Polynomial ($d = 0.40$)	Yes
Mmc	Polynomial ($d = 3.3$)	No	Linear	Yes

Table 4.3: Comparison of Imc, Ajpf and Mmc in Experiment 1.



(a) Linear scale on y-axis

(b) Logarithmic scale on y-axis

Figure 4.7: Verification times of IMC (continuous line, measurements as $\diamond$), AJPF (dashed line, measurements as $\square$), and MMC (dotted line, measurements as $\triangledown$) in Experiment 2. Plotted lines are best-fit regression lines.

AJPF AJPF reports, for all n, that it has visited nine states, i.e. only half the number of states that it explored during Experiment 1. The reason is that AJPF does not need to backtrack and explore other interleavings of the first part of the search, because the first path that AJPF traverses already satisfies $\neg\varphi$ such that violation of φ is established. The first seven states ($state_0$ up to and including $state_6$) are the same as in Example 1. Between $state_6$ and $state_7$, `buggyBlocksAgent`'s goal is removed (but no new action is executed since there are no action options), and during the last transition (to $state_8$), AJPF determines that φ is violated on the current path. It then terminates, and reports a counterexample containing all these nine states.

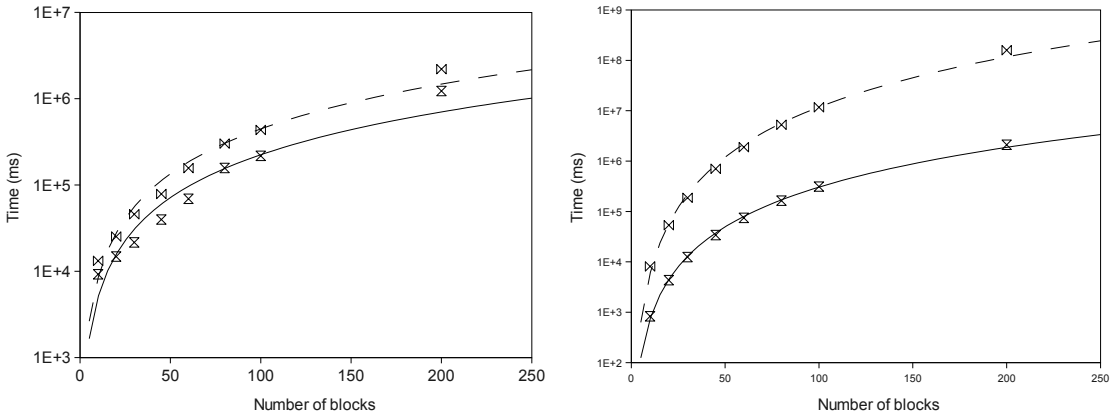
MMC MMC reports, for all n, that it has examined four states; the counterexample contains all of them. The first, second, and fourth state are identical to the states encountered by IMC (excluding the initial source state $\langle \imath_P, \imath_{\neg\varphi} \rangle$). The third state corresponds to the intermediate mental state in which `on(ac,table)` is both belief and goal.

Verification time

The verification times are displayed in Fig. 4.7.

For IMC, the relation between number of blocks and verification time is linear ($R^2 = 0.9615$) or polynomial in degree 0.65 ($R^2 = 0.9500$); we assume the linear relation (similar to the regression analysis of IMC with respect to verification time in Experiment 1). For AJPF, the relation between the number of blocks and verification time is exponential ($R^2 = 0.9422$), or polynomial in degree 1.6 ($R^2 = 0.9406$). Though the difference in R^2 is too slim to be decisive, we are inclined to assume a polynomial relation based on Experiment 1. For MMC, the best-fit function is polynomial in degree 1.7, and the fit is nearly perfect ($R^2 = 0.9988$) similar to Experiment 1. Second-best is a polynomial function ($R^2 = 0.8613$); we assume the polynomial function to describe the relation correctly.

Like in Experiment 1, these results suggest that IMC scales well to larger belief bases, in contrast to AJPF and MMC. Between the latter two, however, there is a big difference as compared to Experiment 1: their verification times are, for the values of n under investigation, much more alike in this experiment. The differences are displayed in Fig. 4.8, which shows that the main reason for the smaller gap is the enormous decrease in verification time of MMC: for $n = 100$, AJPF is almost 2 times faster than in Experiment 1, whereas MMC is 37 times faster. The speed-up of AJPF seems



(a) Ajpf (b) Mmc

Figure 4.8: Comparison of verification times of Ajpf in Experiment 1 (dashed, ⋈) and Experiment 2 (continuous, ⧖), and Mmc in Experiment 1 (dashed, ⋈) and Experiment 2 (continuous, ⧖). Dashed lines are best-fit regression lines of Experiment 1; continuous lines are best-fit regression lines of Experiment 2. Scale on the y-axis is logarithmic in both figures.

to correlate with the number of states it has visited: 18 in Experiment 1 versus 9 in Experiment 2. The speed-up of Mmc, on the other hand, is more difficult to explain, since the number of examined states was equal in both experiments. As the only difference is the substitution of a "normal" transition in Experiment 1 for a "goal-removal"-transition in Experiment 2 (by means of an anonymous action), the decrease in verification time implies that the latter can be performed much faster. We will see later why this is the case.

For $n = 200$, the absolute difference in verification time is the largest: Imc took a second, Ajpf took approximately 20 minutes, and Mmc finished in 35 minutes. This again exemplifies the enormous decrease in verification time for Mmc as compared to Experiment 1: in that experiment, Mmc needed 44 hours to finish model checking.

Memory consumption

The memory consumption is displayed in Fig. 4.9.

For Imc, the best-fit function is a polynomial function in degree 0.37 ($R^2 = 0.8180$), whereas the second-best fit function is logarithmic ($R^2 = 0.7789$); it seems both safe and fair to assume the polynomial relation. For Ajpf, things are more complicated. As can be seen in the figure, the measurements on memory consumption for Ajpf are erratic. This manifests itself in the regression analysis: the best-fit function, a polynomial function in degree 0.23, yielded an R^2 value of only 0.5835. As this is quite low, we do not claim anything about the relation in this case; the only thing that can be observed is that Ajpf requires more memory than the other two model checkers. For Mmc, the analysis is more conclusive. The best-fit function is exponential ($R^2 = 0.9763$), whereas the second-best fit is obtained with a linear function ($R^2 = 0.9584$). Similar to Experiment 1, we favour the linear relation as the radix of the exponential function is again very close to 1, making it behave like a constant. Despite the erratic behaviour of Ajpf, the figures of Experiment 1 and Experiment 2 with regard to memory consumption are quite alike. The absolute difference in memory consumption is the largest for $n = 100$: Imc needed approximately 13 MB, Ajpf required 82 MB, and Mmc occupied 19 MB.

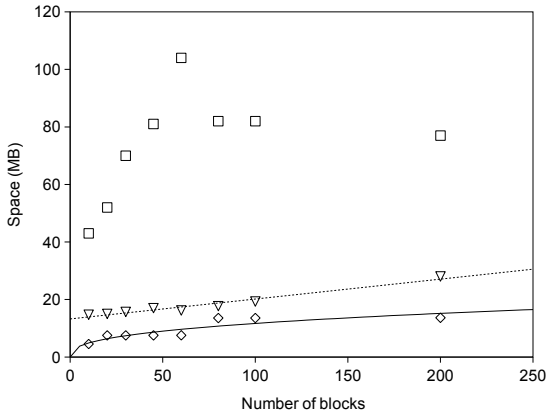


Figure 4.9: Memory consumption of IMC (continuous line, measurements as ◊), AJPF (measurements as □), and MMC (dotted line, measurements as ▽) in Experiment 3. Plotted lines are best-fit regression lines.

Execution

A comparison of resource consumption during verification and normal execution is given in Fig. 4.10.

With respect to runtime, the most interesting readings are generated by MAUDE: its model checker is substantially faster than normal execution: the largest absolute difference is as much as 74 seconds (for $n = 200$). To explain these differences, we need to dive deeper into the workings of MAUDE. In the remainder of this paragraph, we assume basic familiarity with the MAUDE language and its execution environment; otherwise, the MAUDE primer [1] is an excellent introduction.

When model checking GOAL agents with MMC, we load all the required MAUDE modules into its memory, and enter the following command:

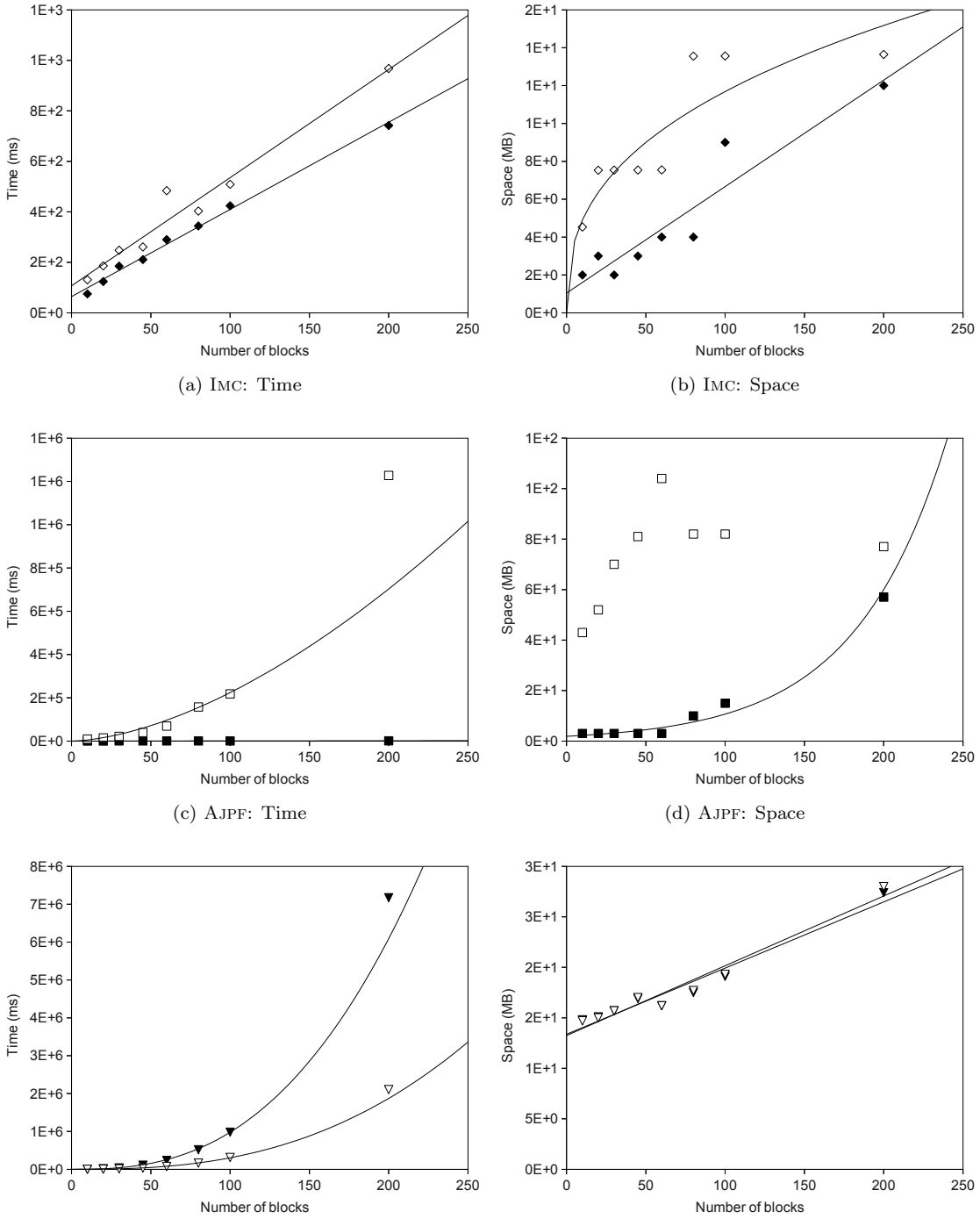
```
Maude> red modelCheck(ms0, <> prop) .
```

In the above, `ms0` represents the initial mental state, and `prop` represents the mental state condition **bel**`(on(ab,table))` (both are implemented as MAUDE equations). In contrast, if we want to execute a GOAL agent, we enter a simple rewrite command:

```
Maude> rew ms0 .
```

Now, the large differences between execution and verification times are due to the different *search regimes* that are applied during model checking and "normal" rewriting. That is, the model checker handles applicable rewrite laws under a depth-first search regime until either a previously encountered state is reached or no more laws are applicable (i.e. in accordance with [27, 60]). However, such a depth-first search cannot be applied to resolve the `rew` command, because a (collection of) rewrite law(s) may be infinitely applicable. In such case, evaluation of the `rew` command would never terminate under a depth-first search regime. Therefore, `rew` commands are resolved with a branching search strategy, in which rewrite laws are scheduled in a round robin fashion.

How does this explain the differences? Well, if the first rewrite path that is traversed by the model checker is already the one leading to a counterexample, then it is reasonable that the round robin search will take a longer time to get there. This is exactly what happens in this experiment: because the anonymous action, performed in the third mental state, is defined *earlier* (as a conditional rewrite law) in the MAUDE module that implements the GOAL semantics than all other actions, it is always the first that is considered by the model checker (rewrite laws are considered in top-to-bottom order).



(a) Imc: Time

(b) Imc: Space

(c) Ajpf: Time

(d) Ajpf: Space

(e) Mmc: Time

(f) Mmc: Space

Figure 4.10: Resource consumption during verification and execution of Imc (measurements as open and closed $\diamond$ for verification and execution, respectively), Ajpf (measurements as open and closed $\square$ for verification and execution, respectively), and Mmc (measurements as open and closed $\triangledown$ for verification and execution, respectively) in Experiment 2. Plotted lines are best-fit regression lines.

Finally, with regard to memory consumption, there appears to be no difference between execution and verification in MAUDE. Although regression analysis was inconclusive for AJPF, it is quite clear that its memory consumption is much higher during verification than during execution. This also holds for IMC.

Intermediate conclusion

The main motivation for Experiment 2 was to be more fair towards AJPF. Since the state space that AJPF needed to explore was a branching structure rather than a single path in Experiment 1, we wanted to see if AJPF would do better (especially with regard to verification time) if it would only have to traverse a single path (like IMC and MMC in Experiment 1). The results show that indeed AJPF is faster: its verification times are roughly halved, bringing AJPF closer to the performance of IMC, although it is still substantially slower.

The big surprise of this experiment, however, is the decrease in verification time of MMC. It has made us speculate about MMC's possible inability to scale to larger state spaces, in addition to its inability to scale to larger belief bases with respect to verification time (though MMC was a lot faster in Experiment 2, it still showed polynomial growth in degree 1.7). Indeed, if the reduction of the state space by a single state decreases MMC's verification times this much, it is interesting to know what will happen if the state space grows larger.

Unfortunately, the regression analysis of memory consumption has been inconclusive for AJPF, although the difference between its demands and the other two model checkers' is, like in Experiment 1, clearly observable. Table 4.4 summarises the regression analyses.

4.3.3 Experiment 3: Counting

To test the hypothesis that MMC does not scale well to larger state spaces, in this experiment, we fix the size of the belief base, while varying the size of the state space. Because `blocksAgent` and `buggyBlocksAgent` are less apt to model these requirements, we introduce a new scenario in which the experimental agent is a simple counter. This agent, called `counterAgent`, starts at 0, and counts until infinity.

There are various ways to implement such counting behaviour in GOAL. We chose an implementation in which:

1. the belief base is used as little as possible, and
2. goals are adopted very frequently.

The reason for limiting belief base queries (i.e. Item 1) is that the effect of the size of the belief base on the performance of the model checkers has already been investigated in the previous experiments; in this experiment, the effect of the size of the state space is under investigation. These two dimensions should be mixed as little as possible. The reason for issuing many goals (i.e. Item 2) is twofold. On the one hand, by relieving the belief base from duty, some other component (i.e. the goal base) should take care of some of the functionality. On the other hand, goal creation is known to be a relatively slow operation in the standard GOAL interpreter. Thus,

Model checker	Verification Time Relation	Scalable	Memory Consumption Relation	Scalable
IMC	Linear	Yes	Polynomial ($d = 0.37$)	Yes
AJPF	Polynomial ($d = 1.6$)	No	–	–
MMC	Polynomial ($d = 1.7$)	No	Linear	Yes

Table 4.4: Comparison of IMC, AJPF and MMC in Experiment 3.

```
main: counterAgent{
  beliefs{
    current(0).
    memory(0).
  }
  goals{
  }
  program{
    if not(goal(current(I))), bel(current(J)), bel(K is J+1)
    then adopt(current(K)).
    %
    if a-goal(current(K)), bel(J is K-1)
    then updateCurrent(J,K).
  }
  actionspec{
    updateCurrent(J,K){
      pre{ true }
      post{ not(current(J)) , current(K) }
    }
  }
}
```

Figure 4.11: Counter agent, called `counterAgent`.

by issuing many goals we provide an additional challenge to Imc, potentially uncovering one of its possible weaknesses.

The agent is given (in standard Goal syntax) in Fig. 4.11. Initially, `counterAgent` has two beliefs: the `current/1` predicate is used to express what its current number is, whereas the `memory/1` predicate may be used to keep track of the numbers that `counterAgent` has in its "memory". In this experiment, the `memory/1` predicate is not yet used (see Experiment 4), except in the initial belief base. Thus, during execution, the belief base contains a single redundant belief.

The counting process works as follows. To get to the next number `counterAgent` first adopts the goal to "compute" the next number (first action rule). Then, in the following reasoning cycle, it updates its belief base accordingly (second action rule). As already briefly mentioned, `counterAgent` can count until infinity. That is, the two action rules can be executed (in alternating order) indefinitely, continuously bringing about new mental states. As a result, the state space of the program automaton is infinite, hence the state space of the product automaton is infinite. To ensure that the model checking procedure eventually terminates, the property φ must be verifiable in a finite number of reasoning cycles; one such property is $\varphi = \mathsf{F}\,\mathbf{bel}(\texttt{current}(n))$ in which $n \in \{10, 20, 30, 45, 60, 80, 100, 200\}$. Because `counterAgent` satisfies φ, eventually $\mathbf{bel}(\texttt{current}(n))$ is be true, making $\neg\varphi$ unsatisfiable on the path that is at that moment explored by the model checker. Since `counterAgent` has only a single computation, the model checker will backtrack all the way to the initial state, and terminate execution thereafter.

State space

In contrast to the state spaces in the previous experiments, the size of the state space in this experiment grows as n in $\varphi = \mathsf{F}\,\mathbf{bel}(\texttt{current}(n))$ becomes larger. However, for all three model checkers, there is a simple formula to compute the size of the state space for any given n.

Imc Imc starts in the product source state, i.e. $\langle \imath_P, \imath_{\neg\varphi} \rangle$. Then, it transits to the product state that is a pair of the initial mental state (in which `current(0)` is a belief) and the property state $q_{\neg\varphi} = \{\neg\mathbf{bel}(\texttt{current}(n)), \mathsf{XG}\neg\mathbf{bel}(\texttt{current}(n))\}$. During the second transition, the goal `current(1)` is adopted, and the third transition achieves this goal by adding `current(1)` to the belief base (and removing `current(0)`). Thus, both "initialisation" (i.e. the product source state and the product state corresponding to the actual initial mental state) and the process of incrementing `current(`i`)` take two product states, e.g. `counterAgent` believes `current(0)` in the second state, it believes `current(1)` in the fourth state, it believes `current(2)` in the sixth state, et cetera. Let $|\mathcal{Q}_{\times}^{\text{Imc}}|$ denote the size of the state space explored by Imc. Then:

$$\begin{aligned} |\mathcal{Q}_{\times}^{\text{IMC}}| &= 2 + 2 \cdot n \\ &= 2 \cdot (n + 1) \end{aligned}$$

AJPF AJPF starts, like IMC, with a dummy initial state. The next four states, similar to Experiment 1 and 2, cover the first reasoning cycle of `counterAgent` (and the empty environment). In this first reasoning cycle, the goal `current(1)` is adopted, *after which reasoning continues*, leading `counterAgent` to the performance of `updateCurrent(0,1)`. Since updates to the mental state (with the exception of **adopt** actions) are carried out at the beginning of the next reasoning cycle, `counterAgent` does not yet believe `current(1)` in the fourth state. Between the fourth and the fifth state, the (again empty) environment is given the opportunity to reason. Subsequently, between the fifth and the sixth state, similar behaviour as between the second and the fourth state is observed: `current(1)` is added to the belief base, the goal `current(1)` is removed, the goal `current(2)` is adopted, and `updateCurrent(1,2)` is said to be performed (but the belief base is, as before, not yet updated). Then, between the sixth and the seventh state, the environment may reason again, et cetera.

Eventually, as soon as `current(`n`)` is in the belief base, AJPF backtracks, and start exploring different interleavings of the first part of the search (similar to Experiment 1). Because belief base updates are carried out during the next reasoning cycle, an additional state (after execution of `updateCurrent(`$n-1, n$`)`) is needed to actually carry out the update. Only then, AJPF will notice the unsatisfiability of $\neg\varphi$ on the current path, and starts backtracking. The exploration of the interleavings at the beginning always (i.e. regardless of the value of n) results in nine new states. Let $|\mathcal{Q}_{\times}^{\text{AJPF}}|$ denote the size of the state space explored by AJPF. Then:

$$\begin{aligned} |\mathcal{Q}_{\times}^{\text{AJPF}}| &= 1 + 4 + 2 \cdot (n-1) + 1 + 9 \\ &= 2 \cdot (n-1) + 15 \\ &= 2 \cdot (n-1) + 2 \cdot 7 + 1 \\ &= 2 \cdot (n+6) + 1 \end{aligned}$$

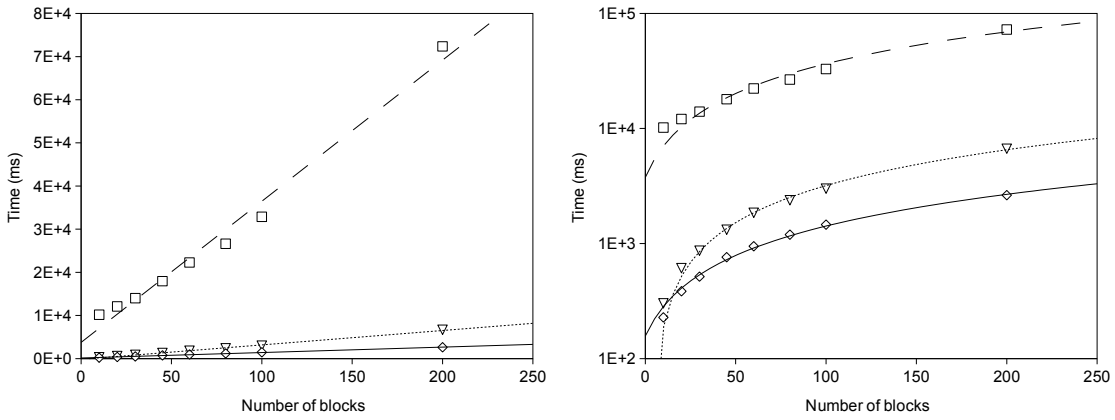
MMC MMC starts with a combination of the initial mental state and the property state $q_{\neg\varphi}$ (identical to the *second* state explored by IMC). To increment the current number, three transitions are needed: the first adopts the goal `current(1)`, the second achieves this goal by adding `current(1)` to the belief base (and removing `current(0)`), and the third removes the goal `current(1)`. In case n is added to the belief base, MMC will backtrack before removing `current(`n`)` from the goal base, as it already can established that $\neg\varphi$ is unsatisfiable on the current path in the intermediate mental state. Let $|\mathcal{Q}_{\times}^{\text{MMC}}|$ denote the size of the state space explored by MMC. Then:

$$\begin{aligned} |\mathcal{Q}_{\times}^{\text{MMC}}| &= 1 + 3 \cdot n - 1 \\ &= 3 \cdot n \end{aligned}$$

Verification time

The verification times are displayed in Fig. 4.12.

For IMC, the relation between the size of the state space and verification time is polynomial in degree 0.82 ($R^2 = 0.9989$) or linear ($R^2 = 0.9974$); as both fits are nearly perfect, in the remainder we assume the worst-case relation, namely linear. For AJPF, the best-fit function is linear ($R^2 = 0.9829$), whereas the second-best fit is obtained when fitting an exponential function ($R^2 = 0.9697$); we assume the linear relation, because the R^2 value of the linear function is higher (though the difference is small), and the radix of the exponential function is close to 1, making it behave like a constant. For MMC, the relation is either polynomial in degree 1.0 ($R^2 = 0.9966$) or linear ($R^2 = 0.9985$). Since a polynomial function in degree 1.0 is in fact a linear function, there is little to discuss here: we assume the linear relation. The largest absolute difference, measured



(a) Linear scale on y-axis

(b) Logarithmic scale on y-axis

Figure 4.12: Verification times of Imc (continuous line, measurements as ⋄), Ajpf (dashed line, measurements as □), and Mmc (dotted line, measurements as ▽) in Experiment 3. Plotted lines are best-fit regression lines.

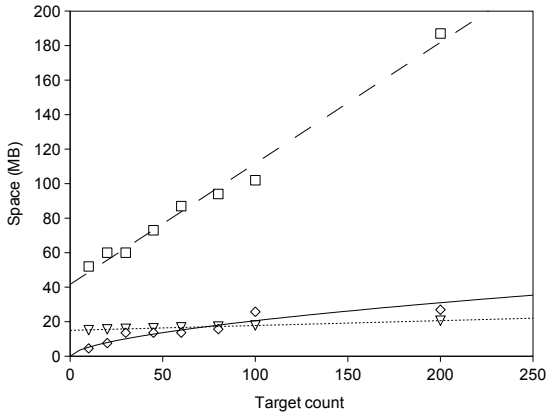


Figure 4.13: Memory consumption of Imc (continuous line, measurements as ⋄), Ajpf (dashed line, measurements as □), and Mmc (dotted line, measurements as ▽) in Experiment 3. Plotted lines are best-fit regression lines.

for $n = 200$, is as follows: Imc took almost 3 seconds, Ajpf took 72 seconds, and Mmc took 7 seconds.

Though we expected Mmc to be the slowest of the three, Ajpf is in fact ten times slower: the slope of the linear function fitted on the Ajpf measurements is roughly 330, whereas the slope of Mmc's linear fit is only 33. Again, this shows that scalability does not guarantee good performance: although the relations are linear for all three model checker (thus implying scalability), there are some clear performance differences.

Memory consumption

The memory consumption is displayed in Fig. 4.13.

Imc For Imc, the function that fitted the measurements on memory consumption best is polynomial in degree 0.59 ($R^2 = 0.9109$), whereas the second-best fit is obtained when fitting a loga-

rithmic function $R^2 = 0.8719$; we assume the polynomial relation. For both AJPF and MMC, the best-fit function is linear ($R^2 = 0.9860$ and $R^2 = 0.9982$, respectively), whereas the second-best fit is an exponential function ($R^2 = 0.9758$ and $R^2 = 0.9942$, respectively) with radix close to 1, thus behaving like a constant. Therefore, we assume the linear relations. With regard to MMC, this implies that its memory consumption grows faster in the size of the state space than that of IMC. Consequently, for some big value of n, IMC will consume less memory than MMC, which is not directly obvious from Fig. 4.13. For $n = 200$, the absolute difference in memory consumption is the largest: IMC required 27 MB, AJPF took 187 MB, and MMC occupied 21 MB.

Execution

A comparison of resource consumption during verification and normal execution is given in Fig. 4.14. With respect to runtime, AJPF stands out, because its verification and execution times differ by much. In contrast, the verification and execution times for IMC and MMC are a lot more alike. This implies that the overhead of model checking with AJPF is substantial, whereas IMC and MMC are relatively efficient in this respect.

With regard to memory consumption, the results are similar to previous experiments: model checking requires a lot more space than normal execution. For MMC, the memory consumption is even constant during normal execution, whereas during verification, a steady increase is measured.

Intermediate conclusion

We started this experiment with the hypothesis that MMC would scale poorly in the size of the state space. This hypothesis was clearly wrong (see also Table 4.5). The experimental results show that MMC scales linear in the size of the state space with regard to both verification time and memory consumption. IMC, like in the previous experiments, is the fastest of the three model checkers, but for some of the values of n under investigation, it consumed more memory than MMC. As outlined, however, it is to be expected that if n grows large enough, MMC will be more memory demanding as the relation for MMC is found to be linear versus a polynomial relation in $d < 1$ for IMC.

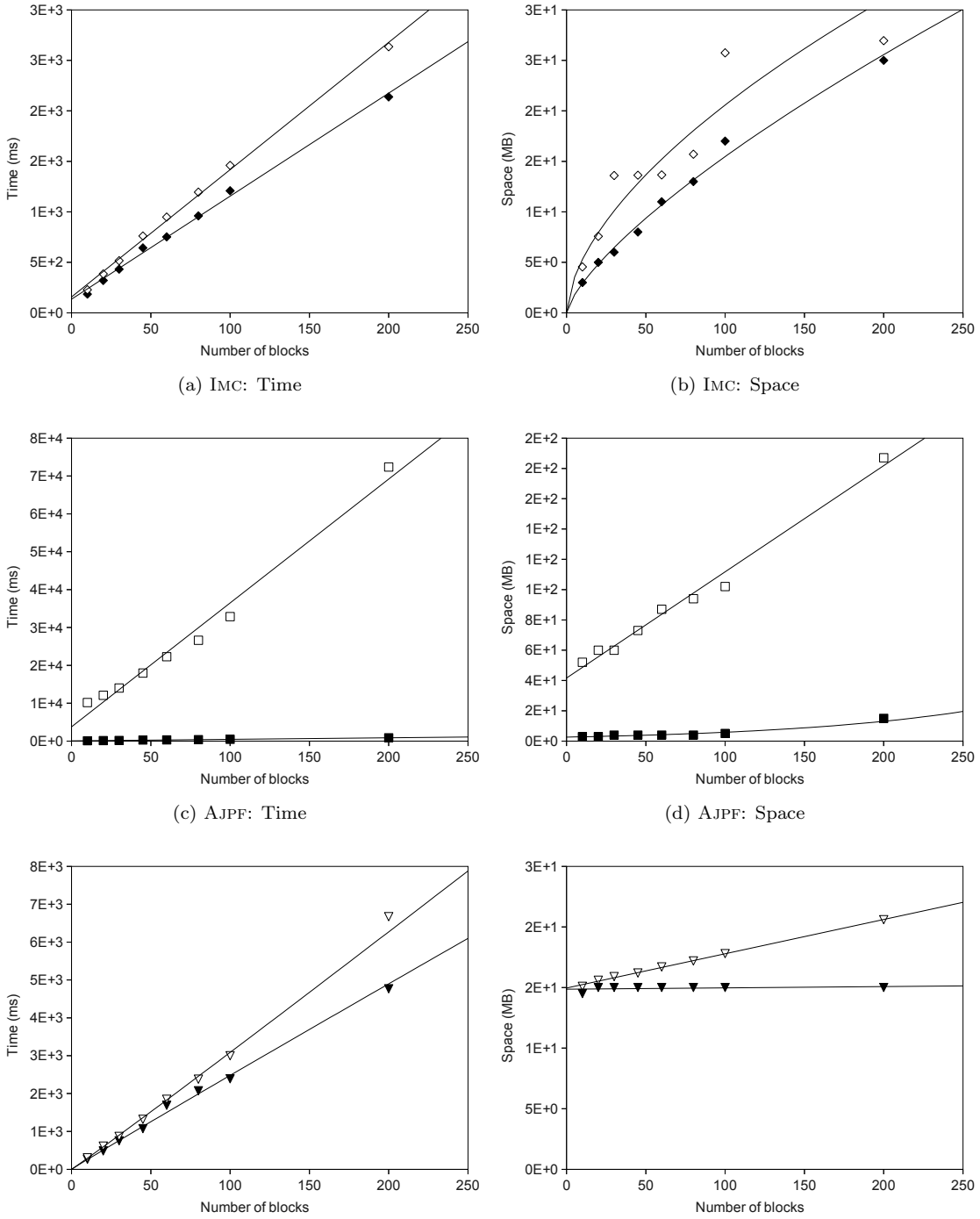
Similar to MMC, AJPF has shown a linear relation between resource consumption and the size of the state space. However, both the intercept and the slope of the linear functions fitted on the AJPF measurements are higher than those of MMC, nor are those functions competitive with the functions fitted on the IMC measurements. In other words, AJPF was the weakest of the three model checkers in this experiment.

4.3.4 Experiment 4: Counting and Memorising

Experiment 3 showed that MMC scales well in the size of the state space. However, the size of the belief base was kept constant in Experiment 3, and belief base queries were issued as little as possible. Experiment 1 and Experiment 2 already showed that MMC does not scale well to larger belief bases; in this last experiment, we adapt `counterAgent` such that it "remembers" all the counted numbers using the aforementioned `memory/1` predicate. Thus, both the size of the belief

Model checker	Verification Time		Memory Consumption	
	Relation	Scalable	Relation	Scalable
IMC	Linear	Yes	Polynomial ($d = 0.59$)	Yes
AJPF	Linear	Yes	Linear	Yes
MMC	Linear	Yes	Linear	Yes

Table 4.5: Comparison of IMC, AJPF and MMC in Experiment 3.



(a) Imc: Time

(b) Imc: Space

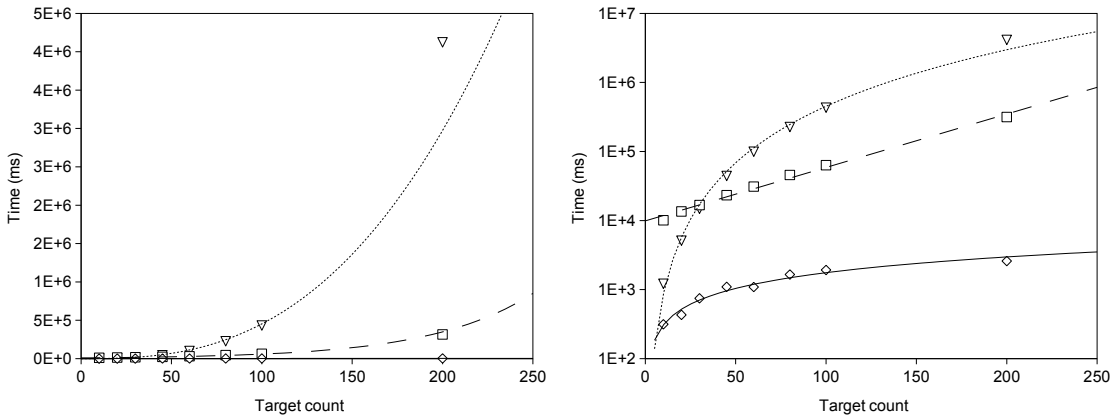
(c) Ajpf: Time

(d) Ajpf: Space

(e) Mmc: Time

(f) Mmc: Space

Figure 4.14: Resource consumption during verification and execution of Imc (measurements as open and closed $\diamond$ for verification and execution, respectively), Ajpf (measurements as open and closed $\square$ for verification and execution, respectively), and Mmc (measurements as open and closed $\triangledown$ for verification and execution, respectively) in Experiment 3. Plotted lines are best-fit regression lines.



(a) Linear scale on y-axis

(b) Logarithmic scale on y-axis

Figure 4.15: Verification times of IMC (continuous line, measurements as ◇), AJPF (dashed line, measurements as □), and MMC (dotted line, measurements as ▽) in Experiment 4. Plotted lines are best-fit regression lines.

base and the state space are varied. To this end, the postcondition of `updateCurrent(J,K)` is changed as follows:

post{ `not(current(J)), current(K), memory(J)` }

Apart from this altered action specification, `counterAgent` is exactly the same as in Fig. 4.11. The size of the belief base now increases linearly in the size of the state space, but note that the `memory` beliefs are never queried: they take no part in any unification process. The property under investigation is, like in Experiment 3, $\varphi = \mathsf{F}\,$**bel**`(current(`n`))` such that all three model checkers terminate eventually.

Because neither the size of the state space, nor the size of the belief base is a constant in this experiment, we will not speak of the relation between resource consumption and either of the two dimensions, but rather about the relation with the "experimental conditions".

State space

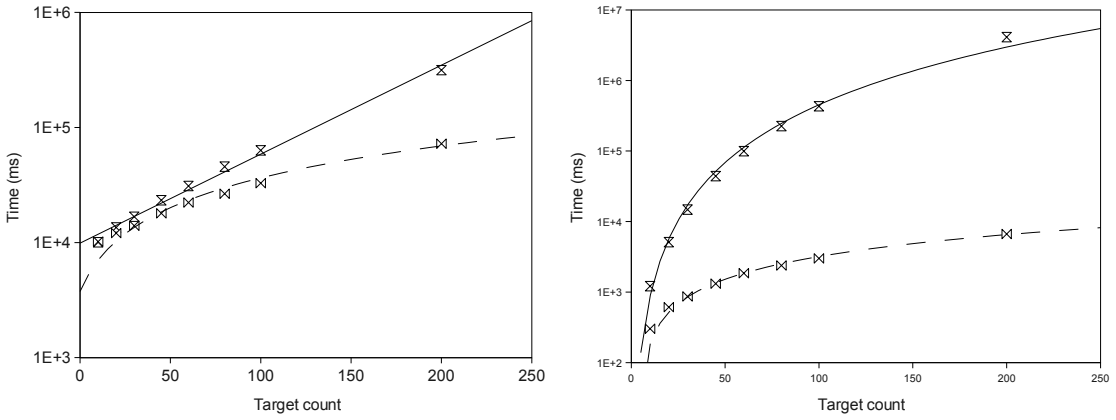
The formulas for computing the size of the state spaces of the model checkers are the same as in Experiment 3.

Verification time

The verification times are displayed in Fig. 4.15.

For IMC, the function that fits the measurements on verification time best is polynomial in degree 0.75 ($R^2 = 0.9722$), whereas the second-best fit is obtained when fitting a logarithmic function ($R^2 = 0.9289$); as before, we assume a linear relation. For AJPF, the best-fit function is exponential ($R^2 = 0.9925$), followed by a linear function ($R^2 = 0.8849$); because the difference in R^2 is large, the relation is assumed exponential. For MMC, a polynomial function in degree 2.7 fits the measurements best ($R^2 = 0.9936$), whereas the second-best fit is exponential ($R^2 = 0.8694$); the polynomial relation is assumed. Note that in neither of the previous experiments, the regression analysis was as conclusive as here. For $n = 200$, the absolute difference in verification time is the largest: IMC finished in 3 seconds, AJPF took 5 minutes, while MMC took over an hour.

The differences in performance of AJPF and MMC with respect to Experiment 3 are very well observable, and displayed in Fig. 4.16. Since this experiment only differs from Experiment 3 in the growing belief base, it seems justified to say that the performance decrease of MMC is all due



(a) Ajpf (b) Mmc

Figure 4.16: Comparison of verification times of Ajpf in Experiment 3 (dashed, ⋈) and Experiment 4 (continuous, ⧖), and Mmc in Experiment 3 (dashed, ⋈) and Experiment 4 (continuous, ⧖). Dashed lines are best-fit regression lines of Experiment 3; continuous lines are best-fit regression lines of Experiment 4. Scale on the y-axis is logarithmic in both figures.

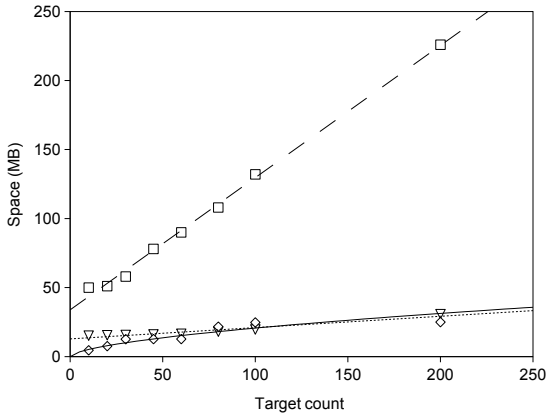


Figure 4.17: Memory consumption of Imc (continuous line, measurements as ⋄), Ajpf (dashed line, measurements as □), and Mmc (dotted line, measurements as ▽) in Experiment 4. Plotted lines are best-fit regression lines.

to this difference. That is, the relation drops from linear to polynomial in degree 2.7. Likewise, for Ajpf, the relation drops from linear to exponential. In contrast, for Imc, the relation goes from linear to polynomial in degree 0.75, which in fact is more efficient (recall, however, that a polynomial function in degree 0.82 fitted equally well in Experiment 3, but that the worst-case function, the linear function, was assumed).

Memory consumption

The memory consumption is displayed in Fig. 4.17.

For Imc, the function that fits the measurements on memory consumption best is polynomial in degree 0.60 ($R^2 = 0.9113$), whereas the second-best fit is obtained when fitting a logarithmic function ($R^2 = 0.8825$). For Ajpf, the best-fit function is linear ($R^2 = 0.9966$, i.e. the fit is nearly perfect), followed by an exponential function ($R^2 = 0.9449$); the linear function is assumed to

describe the relation best. For MMC, the two functions with the highest R^2 values are exponential ($R^2 = 0.9703$) and linear ($R^2 = 0.9491$); because the radix of the exponential function is very close to 1, making it behave like a constant, the linear function is assumed.

The results are very similar to Experiment 3: AJPF performs, though depending linearly on the experimental conditions (thus scalable), the least well of the three model checkers, whereas IMC and MMC perform roughly equal. It is interesting to see that the intersection point of IMC and MMC, mentioned when treating the memory consumption of the model checkers in Experiment 3, is within the range of the values of n under investigation in Experiment 4. That is, for $n = 100$ and $n = 200$, IMC used roughly 25 MB of memory, whereas MMC consumed 20 MB and 31 MB, respectively. With a consumption of 132 MB and 226 MB, respectively, AJPF occupied substantially more space than IMC and MMC.

Execution

A comparison of resource consumption during verification and normal execution is given in Fig. 4.18. The figures are all very similar to Fig. 4.14: verification and execution times of IMC and MMC are quite alike, whereas verification and execution times of AJPF differ substantially. Also, the comparisons with regard to memory consumption are very similar to those in Experiment 3.

Intermediate Conclusion

In Experiment 3, MMC showed that it scales well to larger state spaces. However, Experiment 4 shows that if the belief base grows, MMC's performance quickly degrades, that is, with respect to verification time. This is especially surprising since most beliefs filling the belief base are never queried. With regard to memory consumption, a degradation in performance is not observed. Due to the slow verification of MMC, AJPF is the second-best model checker in this experiment with regard to verification time. Unfortunately, its memory consumption is, like in the previous experiments, the highest. Again, IMC is the best model checker in terms of both verification time and memory consumption: with regard to the former, it is the only model checker that scales well with respect to both types of resource consumption. Fig. 4.6 summarises these findings.

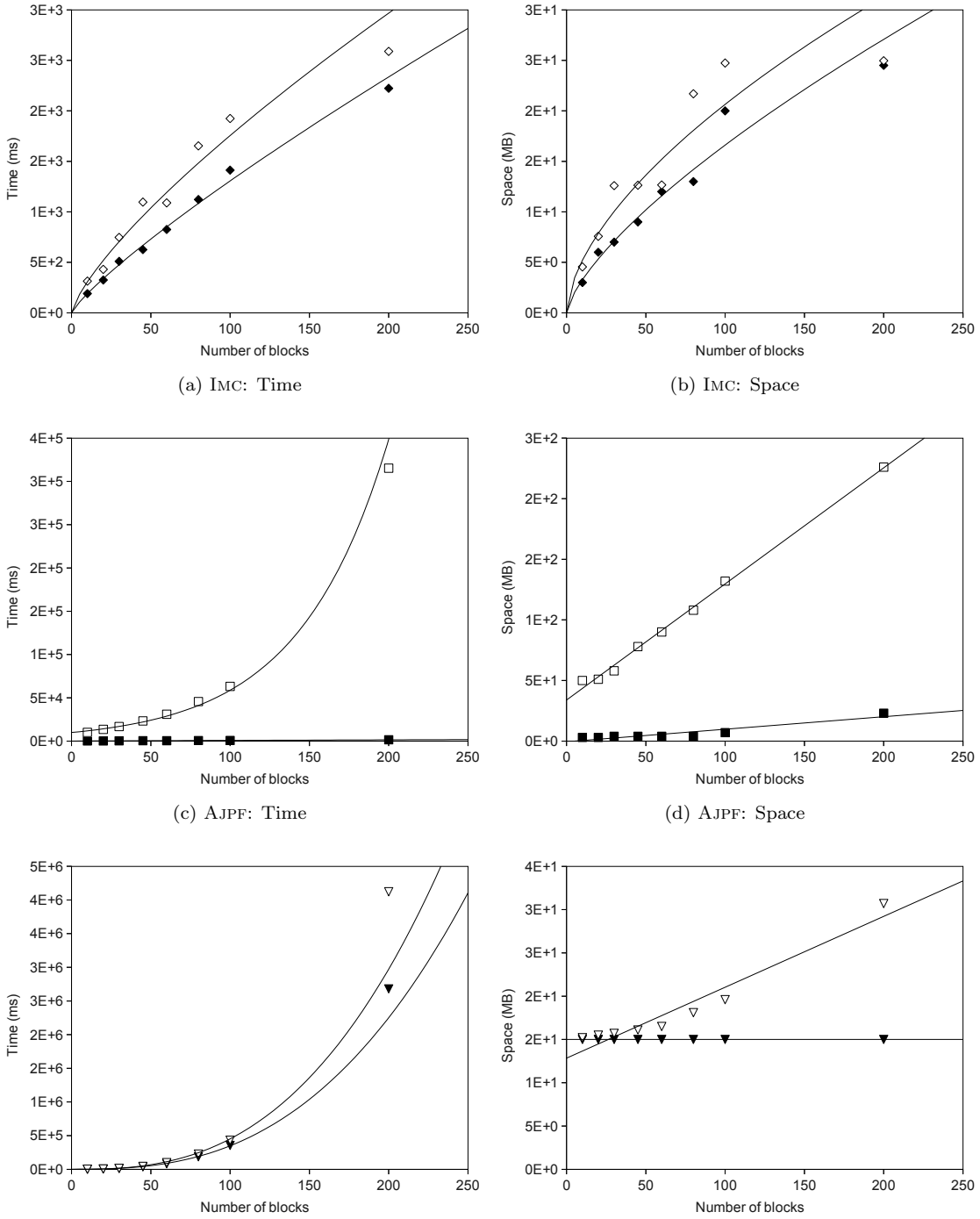
4.4 Discussion

The experimental results, summarised in Table 4.7, clearly show that IMC outperforms the other model checkers. Also, the results show that especially MMC is unable to deal with the simple agents that were under investigation, particularly with regard to verification time; to a lesser extent, this is also true for AJPF.

With regard to AJPF, we believe that for a large part, the overhead of JPF is responsible for the slow verification (as well as for higher memory demands), because executing agents is substantially faster, e.g. the version of `counterAgent` in Experiment 4 easily counts to 200 within a few seconds, whereas verifying with AJPF whether this agent actually can count to 200 takes over five minutes. Similar differences were observed in all experiments. With respect to MMC, the slow verification is partly ascribed to the rate at which the MAUDE GOAL interpreter can generate the state space,

Model checker	Verification time		Memory consumption	
	Relation	Scalable	Relation	Scalable
IMC	Polynomial ($d = 0.75$)	Yes	Polynomial ($d = 0.60$)	Yes
AJPF	Exponential	No	Linear	Yes
MMC	Polynomial ($d = 2.7$)	No	Linear	Yes

Table 4.6: Comparison of IMC, AJPF and MMC in Experiment 4.



(a) Imc: Time (b) Imc: Space

(c) Ajpf: Time (d) Ajpf: Space

(e) Mmc: Time (f) Mmc: Space

Figure 4.18: Resource consumption during verification and execution of Imc (measurements as open and closed $\diamond$ for verification and execution, respectively), Ajpf (measurements as open and closed $\square$ for verification and execution, respectively), and Mmc (measurements as open and closed $\triangledown$ for verification and execution, respectively) in Experiment 4. Plotted lines are best-fit regression lines.

e.g. it took the MAUDE GOAL interpreter already 45 minutes to let `counterAgent` in Experiment 4 count to 200. Note, however, that the overhead of MMC can be substantial as well: verifying whether `counterAgent` actually can count to 200 takes 24 more minutes (in Experiment 4). Note that these two factors in performance degradation correspond to those that identified in Sect. 4.1.

Earlier, we mentioned that model checking non-deterministic agents is infeasible with AJPF and MMC. We carried out an additional "proof-of-concept" experiment with IMC featuring a non-deterministic agent to illustrate this. Although this in principle does not say anything about AJPF and MMC, we believe it does show that the odds are not in their favour. Consider a blocks world agent as in Experiment 1 with an initial belief base containing 200 blocks divided over two towers of 100 blocks each, and a property specifying that the target configuration (all blocks on the table) is reached. This non-deterministic agent has a state space of size 10,000. Nevertheless, verification with IMC takes only 2150 seconds, i.e. 35 minutes. To see this in perspective, note that AJPF and MMC required already more time (40 minutes and 44 hours, respectively) to complete verification for $n = 200$ in Experiment 1: a comparable setting with only four states instead of 10,000. Given that the size of the belief base is constant, and assuming that AJPF and MMC scale linearly in the size of the state space (as suggested by the results of Experiment 2), it would take AJPF 100,000 minutes, i.e. 70 days, and MMC 110,000 hours, i.e. 12.5 *years*, to terminate. Clearly, this is not tractable.

We used regression analysis as a tool to formalise the performance of the model checkers instead of only comparing absolute results. This gave us the ability to make statistically grounded claims about scalability of the model checkers involved. However, we also saw that scalability does not imply well performance. Regression analyses should, therefore, only be used as supporting evidence for claims about model checking performance.

4.5 Summary

In this chapter, we have compared our interpreter-based model checker for GOAL with other efforts in agent verification. We started with an overview of related work in the field of APL model checking. During this exposition, we listed three benefits of our approach over existing approaches: improved performance, increased expressiveness of properties, and immediate / encoding-less language support. The remainder of the chapter has focussed on the first of these claimed advantageous with a quantitative performance analysis of our model checker and two other model checkers

Model checker	Experiment	Verification time		Memory consumption	
		Relation	Scalable	Relation	Scalable
IMC	Experiment 1	Linear	Yes	Polyn. ($d = 0.40$)	Yes
	Experiment 2	Linear	Yes	Polyn. ($d = 0.37$)	Yes
	Experiment 3	Linear	Yes	Polyn. ($d = 0.59$)	Yes
	Experiment 4	Polyn. ($d = 0.75$)	Yes	Polyn. ($d = 0.60$)	Yes
AJPF	Experiment 1	Polyn. ($d = 1.7$)	No	Polyn. ($d = 0.40$)	Yes
	Experiment 2	Polyn. ($d = 1.6$)	No	–	–
	Experiment 3	Linear	Yes	Linear	Yes
	Experiment 4	Polyn.	No	Linear	Yes
MMC	Experiment 1	Polyn. ($d = 3.3$)	No	Linear	Yes
	Experiment 2	Polyn. ($d = 1.7$)	No	Linear	Yes
	Experiment 3	Linear	Yes	Linear	Yes
	Experiment 4	Polyn. ($d = 2.7$)	No	Linear	Yes

Table 4.7: Relations between resource consumption and experimental conditions for IMC, AJPF and MMC in all experiments; Polyn. is an abbreviation for polynomial.

for Goal. The results indeed show that our approach is a lot more efficient. Along the way, we have proposed regression analysis as a statistical tool to assess scalability of model checkers as well as criteria that state when a model checker scales well in experimental conditions. To the best of our knowledge, no other investigations of model checking performance at a similar level of thoroughness exist in the agent verification literature.

Part II

State Space Reduction

Chapter 5

Transition Theory

In this second part of the thesis, we will discuss two optimisation techniques that extend the previously introduced core model checker: *property-based slicing* (PBS) and *partial order reduction* (POR). Before we can treat those, however, additional theory about the execution of a GOAL agent is required. More specifically, in this chapter, we revisit the transition system of a GOAL agent, and introduce a different way of looking at the individual transitions it contains.

In the transition system of a GOAL agent, a transition is a connection between a *source* mental state and a *destination* mental state. In this chapter, we will regard transitions as operations that add and delete belief and goals. Applied to a source, such an operation transforms this mental state to a destination according to the additions and deletions by which the operation is characterised. Consequently, distinct transitions in the transition system that bring about the same change can be thought of as applying the same operation on different source mental states. This operational reading of transitions is used by PBS and POR algorithms in later chapters, e.g. to determine that the order in which two operations are executed does not matter. In such case, only one order need be investigated instead of all orders.

We proceed as follows. In Sect. 5.1, we further refine our notion of the operation perspective on transitions as outlined above. Specifically, we redefine the term "transition" to what we formerly called "operation" to comply to POR literature. Subsequently, in Sects. 5.2 and 5.3, we discuss *read sets* and *write sets*. Both are sets of mental state conditions: the former contains the MSCs that are evaluated to generate a transition, whereas the latter contains the MSCs whose truth value changes due to a transition's execution. Based on these sets, we introduce three relations on transitions in Sect. 5.4 that are used by the POR and PBS algorithms of later chapters. The chapter is concluded in Sect. 5.5. To get a better feel for the concepts we introduce, we illustrate them with examples, i.e. `blenderAgent` reappears. Recall that `blenderAgent`, introduced in Chap. 2, is an agent that puts fruit into a blender to make juice. Its source code is given in Fig. 2.1, and its transition system is shown in Fig. 2.2.

5.1 Transitions

Let us start with refining our notion of transitions. Recall that we defined the transition system $\mathcal{T}$ of a GOAL program P as the tuple $\mathcal{T} = \langle \Omega_M, \mu_0, \longrightarrow \rangle$, in which Ω_M is the set of all mental states of P, $\mu_0 \in \Omega_M$ is the initial mental state, and $\longrightarrow \subseteq \Omega_M \times \Omega_M$ is the transition relation connecting mental states.

In this setting, the notion of transitions describes how a mental state can change to the next, which corresponds to one computation step. In the context of POR, it is useful to be able to refer to the changes (to the belief base and goal base) that the various computation steps bring about. These changes can be viewed as operations that add and delete elements from the belief and goal base, making it easy to express that the order in which two operations are performed is irrelevant (an important property relevant in POR methods). In GOAL, the same operation (i.e. change to belief and goal base) can be caused by the execution of different actions. However, for POR we can

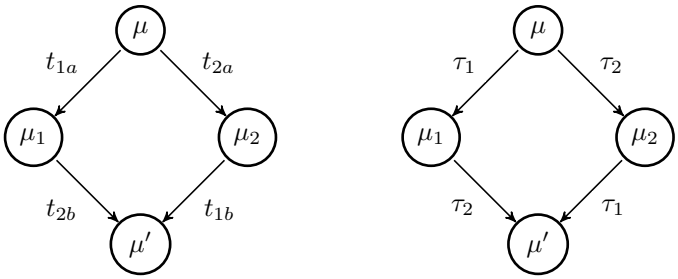


Figure 5.1: Simple transition system.

abstract from the particular action that caused the change, and only consider the actual change itself.

Technically, this gives rise to a different notion of transition. To prevent confusion with the common notion of transitions, we refer to pairs of mental states in the transition system as *steps*, denoted by t, in the remainder of this and subsequent chapters. The new notion of transition is one in which a transition, denoted by τ, consists of a set of steps that bring about the same change to the belief base and goal base. The change associated with such a transition can be viewed as an operation that can be performed on a belief base and goal base in those mental states μ for which a step $t = \langle \mu, \mu' \rangle$ is included in the transition, i.e. the transition is enabled only in those states (see below for a formal definition of when a transition is enabled). This notion of transition as a set of steps corresponds to the notion of transition in the POR literature (e.g. [24]).

Figure 5.1 illustrates the concept of transitions. In the left figure, the transition system is shown with edges annotated with symbolic names for steps. On the right, the same transition system is shown with edges annotated with symbolic names for transitions, i.e. $\tau_1 = \{t_{1a}, t_{1b}\}$ and $\tau_2 = \{t_{2a}, t_{2b}\}$. When viewing the transitions as operations, we can regard τ_1 as a partial function that takes μ as input, and transforms it to μ_1 by adding and deleting beliefs and goals to and from μ. Similarly, we can apply τ_1 on μ_2 resulting in μ' (by adding and deleting the same beliefs and goals as when τ_1 is applied to μ).

Thus, a transition is characterised by the series of additions to and deletions from the belief base and goal base that it brings about. We formalise this as follows. Let the *add/del function* $\Delta : (\Omega_M \times \Omega_M) \to (2^{\mathcal{L}} \times 2^{\mathcal{L}}) \times (2^{\mathcal{L}} \times 2^{\mathcal{L}})$ map a mental state pair $\langle \mu, \mu' \rangle$ to the Add*-pair* $\langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma \rangle$ containing the beliefs (set of facts) and goals (set of conjunctions of facts) that need be added to μ to get to μ', and the Del*-pair* $\langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma \rangle$ containing the beliefs and goals that need be deleted from μ to get to μ'.

Definition 5. *Let $\langle \mu, \mu' \rangle$ be a mental state pair in which $\mu = \langle K, \Sigma, \Gamma \rangle$ and $\mu' = \langle K, \Sigma', \Gamma' \rangle$. Then:*

$$\Delta(\langle \mu, \mu' \rangle) = \langle \langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma \rangle, \langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma \rangle \rangle \text{ in which } \mathsf{Add}_\Sigma = \Sigma' \setminus \Sigma \text{ and } \mathsf{Add}_\Gamma = \Gamma' \setminus \Gamma \text{ and } \mathsf{Del}_\Sigma = \Sigma \setminus \Sigma' \text{ and } \mathsf{Del}_\Gamma = \Gamma \setminus \Gamma'$$

As a less abstract example than Fig. 5.1, consider Fig. 5.2 in which `blenderAgent`'s transition system is displayed, annotated with symbolic names for steps (the transition system originally appeared in Fig. 2.2). Additionally, Table 5.1 lists the 27 steps that appear in the transition system, together with the Add-pairs and Del-pairs resulting from applying Δ to them; we refer to Fig. 2.2 in Chap. 2 for the contents of every mental state in the transition system. For example, suppose we apply Δ to the step $t_0 = \langle \mu_0, \mu_1 \rangle$. The goal base is the same in μ_0 and μ_1, i.e. $\mathsf{Add}_\Gamma = \mathsf{Del}_\Gamma = \emptyset$ (see Fig. 2.2). In contrast, the belief base undergoes change: the PL fact `washed` appears in the belief base of μ_1, whereas it is absent in the initial belief base (again, see Fig. 2.2). As this is the only difference, $\mathsf{Add}_\Sigma = \{\texttt{washed}\}$ and $\mathsf{Del}_\Sigma = \emptyset$. Thus, $\Delta(t_0) = \langle \langle \{\texttt{washed}\}, \emptyset \rangle, \langle \emptyset, \emptyset \rangle \rangle$.

In addition to the change that it brings about, we associate every transition with a unique action rule as well. As such, transitions are connected to the source code by which a GOAL agent is defined, which helps us *compute* the concepts that we introduce in this and the next section (we elaborate on these implementation details in Sect. 5.3). As such, rather than being a set of steps, a transition is a pair of an action rule and a set of steps. This works as follows. A step $t = \langle \mu, \mu' \rangle$ exists by virtue of *at least* one action option in μ that brings about change which results in μ' (we emphasise "at least" because there may exist different action options in μ that bring about exactly the same change). The set of all action rules that may have been responsible for opting the actions by whose virtue t exists is denoted by $\mathsf{Rules}(t)$.

Definition 6. *Let R be the set of action rules, and let $t = \langle \mu, \mu' \rangle$ be a step. Then:*

$$\mathsf{Rules}(t) \quad = \quad \{\textbf{if } \psi \textbf{ then } \alpha \in R \mid \mu \models_\Psi \psi \text{ and } \mu' = \mathcal{M}(\alpha, \mu) \text{ is defined}\}$$

We say that an action option "by whose virtue a step exists" *generates* this step. Similarly, we say that an action rule "that may be responsible for opting an action by whose virtue a step exists" *generates* this step.

Given the definitions of Δ and Rules, we proceed with the definition of a transition.

Definition 7. *Let ρ be an action rule. A transition is a pair $\langle \rho, S \rangle$ in which $S \subseteq \longrightarrow$ is the largest set of steps such that for all $t, t' \in S$: $\Delta(t) = \Delta(t')$ and $\rho \in \mathsf{Rules}(t)$ and $\rho \in \mathsf{Rules}(t')$*

Thus, every transition τ is a pair of an action rule and the largest subset of $\longrightarrow$ containing steps that bring about the same change and can be generated by the action rule. For ease of notation, however, in the remainder we allow ourselves to treat a transition τ as the set of steps it contains (such that we may write "there exists a $t \in \tau$ such that..." instead of "let $\tau = \langle \rho, S \rangle$ such that there exists a $t \in S$ such that..."). Additionally, we refer to the action rule associated with τ by $\mathsf{Rule}(\tau)$, i.e. if $\tau = \langle \rho, S \rangle$ then $\mathsf{Rule}(\tau) = \rho$. We elaborate on the exact reason for incorporating action rules in the definition of a transition in Sect. 5.2.1, after the introduction of some additional notions related to transitions.

To exemplify transitions, we again consider `blenderAgent`'s transition system. In Fig. 5.3, the transition system is shown, this time annotated with symbolic names for transitions. Additionally, the first column of Table 5.2 lists the 9 transitions that can be derived from Table 5.1; together with that table, it is quite easy to see that the change that the steps belonging to the same transition bring about is the same. The second column of Table 5.2 shows the action rule associated with each transition.

We adopt the following notation and terminology.

Universe We denote the set of all transitions of an agent (i.e. its *universe* of transitions) by Ω_T.

Enabled In accordance with the literature on POR (e.g. [24]), a transition τ is said to be *enabled* in a mental state μ, if there exists a step $t \in \tau$ such that $t = \langle \mu, \mu' \rangle$. We denote the set of all enabled transitions in a mental state μ by $\mathsf{En}(\mu)$, i.e. $\mathsf{En}(\mu) = \{\tau \in \Omega_T \mid \text{there exists a } \langle \mu, \mu' \rangle \in \tau\}$.

Successor If $\langle \mu, \mu' \rangle \in \tau$, then, for notational convenience, we denote μ's successor μ' by $\tau(\mu)$ as if τ is a function (e.g. [24]). This is justified by the definition of τ: there exists at most one μ such that $\langle \mu, \mu' \rangle \in \tau$ for all $\mu \in \Omega_M$ (bringing about the same change in the same mental state always results in the same successor).

5.2 Read / Write Sets

In this section, we introduce two concepts, read sets and write sets, that we use to define various relations on transitions in the next subsection. More specifically, both read sets and write sets are sets of mental state conditions that are used to express how enabledness of transitions depends

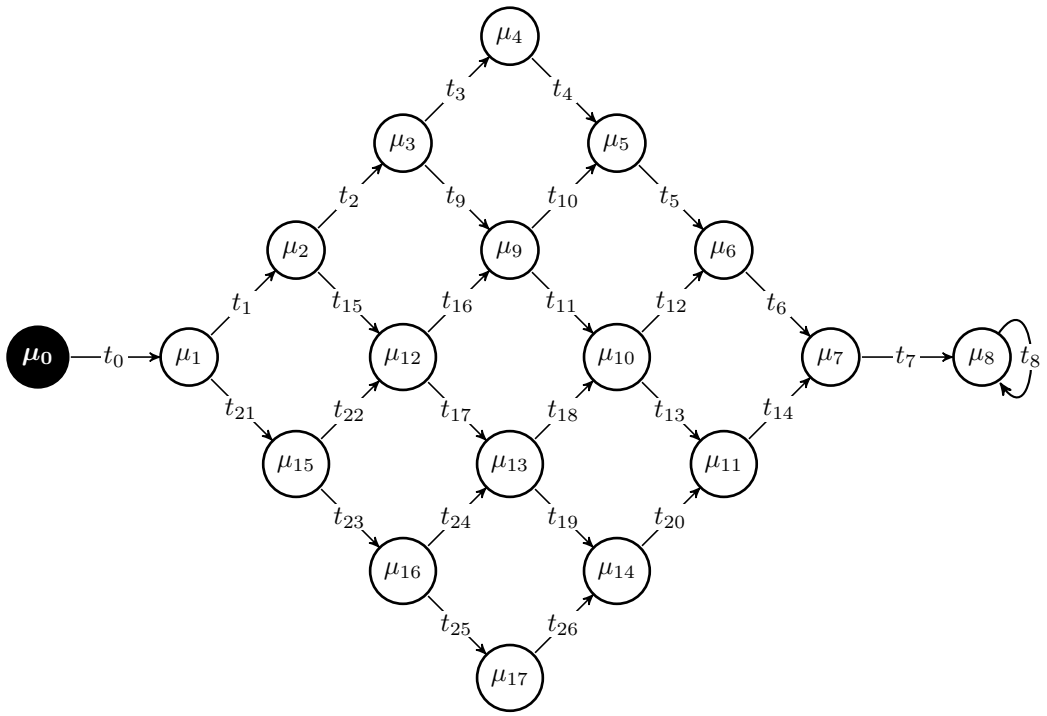


Figure 5.2: Transition system of `blenderAgent`, annotated with symbolic names for steps.

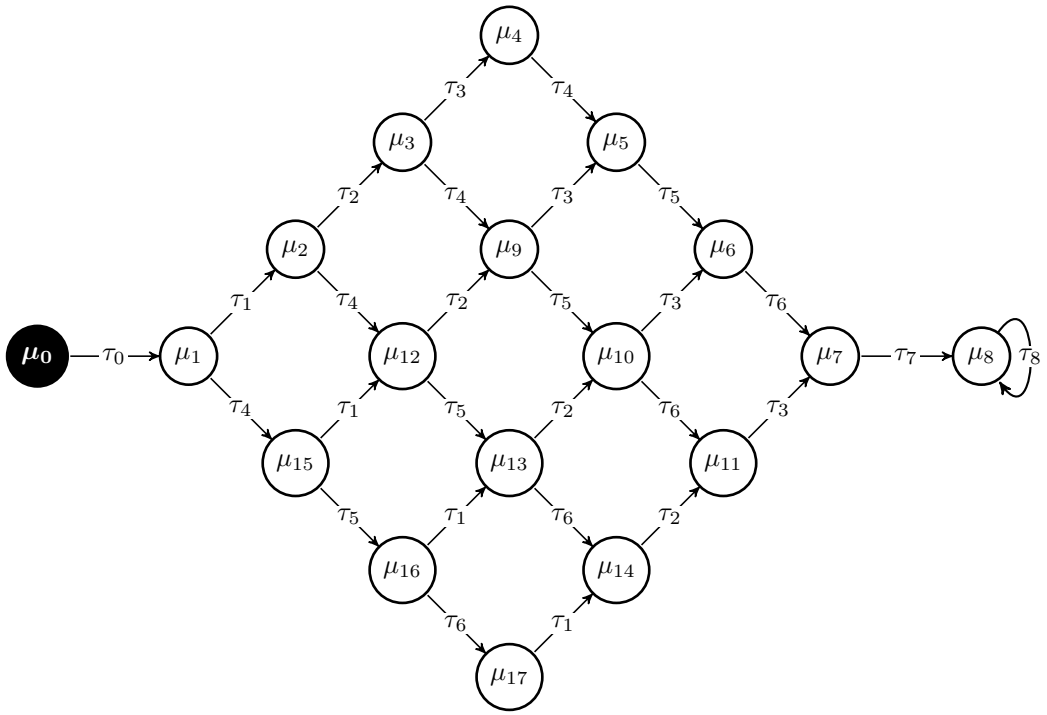


Figure 5.3: Transition system of `blenderAgent`, annotated with symbolic names for transitions.

Step	Add_Σ	Add_Γ	Del_Σ	Del_Γ
t_0	{washed}	∅	∅	∅
t_1	{added(bananas,1)}	∅	{added(bananas,0)}	∅
t_2	{added(bananas,2)}	∅	{added(bananas,1)}	∅
t_3	{ticked(bananas)}	∅	∅	∅
t_4	{added(oranges,1)}	∅	{added(oranges,0)}	∅
t_5	{added(oranges,2)}	∅	{added(oranges,1)}	∅
t_6	{ticked(oranges)}	∅	∅	∅
t_7	{switch(on)}	∅	{switch(off)}	{{filled, switch(on)}}
t_8	∅	∅	∅	∅
t_9	{added(oranges,1)}	∅	{added(oranges,0)}	∅
t_{10}	{ticked(bananas)}	∅	∅	∅
t_{11}	{added(oranges,2)}	∅	{added(oranges,1)}	∅
t_{12}	{ticked(bananas)}	∅	∅	∅
t_{13}	{ticked(oranges)}	∅	∅	∅
t_{14}	{ticked(bananas)}	∅	∅	∅
t_{15}	{added(oranges,1)}	∅	{added(oranges,0)}	∅
t_{16}	{added(bananas,2)}	∅	{added(bananas,1)}	∅
t_{17}	{added(oranges,2)}	∅	{added(oranges,1)}	∅
t_{18}	{added(bananas,2)}	∅	{added(bananas,1)}	∅
t_{19}	{ticked(oranges)}	∅	∅	∅
t_{20}	{added(bananas,2)}	∅	{added(bananas,1)}	∅
t_{21}	{added(oranges,1)}	∅	{added(oranges,0)}	∅
t_{22}	{added(bananas,1)}	∅	{added(bananas,0)}	∅
t_{23}	{added(oranges,2)}	∅	{added(oranges,1)}	∅
t_{24}	{added(bananas,1)}	∅	{added(bananas,0)}	∅
t_{25}	{ticked(oranges)}	∅	∅	∅
t_{26}	{added(bananas,1)}	∅	{added(bananas,0)}	∅

Table 5.1: Steps that occur in the transition system of `blenderAgent`, and the associated Add-pairs and Del-pairs resulting from applying Δ, i.e. $\Delta(t_i) = \langle\langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma\rangle, \langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma\rangle\rangle$ for all $0 \leq i \leq 26$.

Transition	Rule
$\tau_0 = \{t_0\}$	**if goal**(filled) ∧ **bel**(not(washed)) **then** wash
$\tau_1 = \{t_1, t_{22}, t_{24}, t_{26}\}$	**if goal**(filled) ∧ **bel**(toAdd(bananas)) **then** add(bananas,1)
$\tau_2 = \{t_2, t_{16}, t_{18}, t_{20}\}$	**if goal**(filled) ∧ **bel**(toAdd(bananas)) **then** add(bananas,1)
$\tau_3 = \{t_3, t_{10}, t_{12}, t_{14}\}$	**if goal**(filled) ∧ **bel**(recipe(F,_)) **then** tick(F)
$\tau_4 = \{t_4, t_9, t_{15}, t_{21}\}$	**if goal**(filled) ∧ **bel**(toAdd(oranges)) **then** add(oranges,1)
$\tau_5 = \{t_5, t_{11}, t_{17}, t_{23}\}$	**if goal**(filled) ∧ **bel**(toAdd(oranges)) **then** add(oranges,1)
$\tau_6 = \{t_6, t_{13}, t_{19}, t_{25}\}$	**if goal**(filled) ∧ **bel**(recipe(F,_)) **then** tick(F)
$\tau_7 = \{t_7\}$	**if goal**(switch(on)) ∧ **bel**(filled) **then** blend
$\tau_8 = \{t_8\}$	–

Table 5.2: Transitions that occur in the transition system of `blenderAgent`, and the associated action rules. The "–" in the Rule cell of τ_8 is there because τ_8 is not generated by an action rule, but by the absence of applicable action rules (in which case **skip** is performed indefinitely).

on the execution of transitions. Additionally, write sets are also used to determine how execution of transitions can influence the truth value of an LTL formula. We treat both purposes in more detail in Sect. 5.4; first, the theory.

5.2.1 Read set

The mental state conditions that may have been evaluated during the generation of a step in a transition τ are collectively called τ's *read set*, denoted $\mathsf{Read}(\tau)$. Formally, the read set of τ is defined as follows.

Definition 8. *Let τ be a transition, and let $\mathsf{Rule}(\tau) = \textbf{if } \psi \textbf{ then } \alpha$.*
Then:

$$\mathsf{Read}(\tau) \quad = \quad \begin{cases} \{\psi, \textbf{bel}(\chi_{\mathtt{pre}})\} & \text{if } \alpha = \langle \chi_{\mathtt{pre}}, \chi_{\mathtt{post}} \rangle \\ \{\psi, \neg\textbf{bel}(\chi)\} & \text{if } \alpha = \textbf{adopt}(\chi) \\ \{\psi\} & \text{if } \alpha = \textbf{drop}(\chi) \end{cases}$$

Note that we do not have to define Read for the cases when $\alpha \in \{\textbf{insert}(\chi), \textbf{delete}(\chi), \textbf{skip}\}$ as these actions are abbreviations for specific user-defined actions (see Sect. 2.4)

For the definition of the read set as given in Def. 8, it is important that every transition is associated with a single action rule. Otherwise, i.e. if the only requirement for the steps belonging to a transition would be that they bring about the same change, different action rules might be responsible for generating steps belonging to the same transition.[1] Suppose we denote this set of action rules by $\mathsf{Rules}(\tau)$.[2] Then, because we need to account for each step belonging to τ, the definition of Read would look as follows.[3]

$$\mathsf{Read}(\tau) \quad = \quad \bigcup_{\textbf{if } \psi \textbf{ then } \alpha \in \mathsf{Rules}(\tau)} \left(\begin{cases} \{\psi, \textbf{bel}(\chi_{\mathtt{pre}})\} & \text{if } \alpha = \langle \chi_{\mathtt{pre}}, \chi_{\mathtt{post}} \rangle \\ \{\psi, \neg\textbf{bel}(\chi)\} & \text{if } \alpha = \textbf{adopt}(\chi) \\ \{\psi\} & \text{if } \alpha = \textbf{drop}(\chi) \end{cases} \right)$$

This alternative definition makes it impossible to properly extend the notion of read sets to action rules, which we will do in Sect. 5.4. Informally, when transitions are associated with a single rule, if ρ is an action rule and τ can be generated by ρ (i.e. $\mathsf{Rule}(\tau) = \rho$), then we will define a read set for action rules such that $\mathsf{Read}(\rho) = \mathsf{Read}(\tau) = \mathsf{Read}(\mathsf{Rule}(\tau))$.[4] In contrast, when transitions are not associated with a single action rule (i.e. $\rho \in \mathsf{Rules}(\tau)$ rather than $\rho = \mathsf{Rule}(\tau)$) and the alternative definition of a read set need be used, the previous equality cannot hold because ρ does not contain information about possible other action rules in $\mathsf{Rules}(\tau)$. Consequently, only $\mathsf{Read}(\rho) \subseteq \mathsf{Read}(\tau)$ holds in such a case.[5] This means that, given an action rule ρ, we cannot generalise over the mental state conditions that are evaluated to generate the transitions associated with ρ, because these transition may be generated by other action rules than ρ as well. Such generalisations are, however, important, because at the time that we compute read sets, which is before generation of the transition system as we will see, we only have knowledge about action rules (and not about individual transitions because the transition system has not been generated yet). Still, at that

[1] To exemplify such a situation, suppose we have two action rules $\rho_1 = \textbf{if bel(foo) then insert(baz)}$ and $\rho_2 = \textbf{if bel(bar) then insert(baz)}$. Additionally, suppose the current mental state is $\mu = \langle K, \Sigma, \Gamma \rangle$ in which $\Sigma = \{\mathtt{foo}, \mathtt{bar}\}$ and $\Gamma = \emptyset$. Then, both action rules generate the same option in μ, namely $\textbf{insert(baz)}$, hence the same step $t = \langle \mu, \mu' \rangle$ in which $\mu' = \langle K, \{\mathtt{foo}, \mathtt{bar}, \mathtt{baz}\}, \Gamma \rangle$. When transitions are not associated with action rules, this yields one transition $\tau = \{t\}$. When transitions are associated with action rules, this yields two transitions, namely $\tau_1 = \langle \rho_1, \{t\} \rangle$ and $\tau_2 = \langle \rho_2, \{t\} \rangle$.

[2] Continuing the example of the previous footnote, we would have $\mathsf{Rules}(\tau) = \{\rho_1, \rho_2\}$ versus $\mathsf{Rule}(\tau_1) = \rho_1$ and $\mathsf{Rule}(\tau_2) = \rho_2$.

[3] Continuing the example of the previous footnote, we would have $\mathsf{Read}(\tau) = \{\textbf{bel(foo)}, \textbf{bel(bar)}\}$ versus $\mathsf{Read}(\tau_1) = \{\textbf{bel(foo)}\}$ and $\mathsf{Read}(\tau_2) = \{\textbf{bel(bar)}\}$.

[4] Continuing the example of the previous footnote, $\rho_1 = \textbf{if bel(foo) then insert(baz)}$ provides sufficient information to define $\mathsf{Read}(\rho_1) = \{\textbf{bel(foo)}\}$ such that $\mathsf{Read}(\tau_1) = \mathsf{Read}(\rho_1)$. The same holds for τ_2 and ρ_2.

[5] Continuing the example of the previous footnote, we have $\mathsf{Read}(\rho_1) \subseteq \mathsf{Read}(\tau)$, but not $\mathsf{Read}(\rho_1) = \mathsf{Read}(\tau)$ (recall that $\mathsf{Read}(\tau) = \{\textbf{bel(foo)}, \textbf{bel(bar)}\}$). The same holds for τ_2.

time, we already want to make statements about characteristics of transitions generated by the same action rule, e.g. their enabledness. Such analyses can only be done if we can generalise about read sets.

To exemplify read sets (according to Def. 8), consider τ_7 of `blenderAgent`. By Table 5.2, we know that τ_7 can only be generated by the rule **if goal**`(switch(on))` $\wedge$ **bel**`(filled)` **then** `blend` such that **goal**`(switch(on))` $\wedge$ **bel**`(filled)` is in τ_7's read set. Additionally, the precondition of the user-defined action `blend` is `switch(off)` such that **bel**`(switch(off))` is also in τ_7's read set. Thus, the read set of τ_7 is $\mathsf{Read}(\tau_7) = \{$**goal**`(filled)` $\wedge$ **bel**`(filled)`, **bel**`(switch(off))` $\}$. The second column of Table 5.3 shows the read sets for all `blenderAgent`'s transitions; the other columns of this table (as well as the definition of Ψ below the table) are discussed in Sect. 5.2.2.

The purpose of read sets is to formalise on which mental state conditions enabledness of a transition depends. This is an important notion, because if we know that some transition τ_1 is enabled in a mental state μ, and another transition, say τ_2, is also enabled in μ and execution of τ_2 does *not* change the truth values of any of the MSCs in $\mathsf{Read}(\tau_1)$, then we know that if we execute τ_2 in μ, τ_1 does not become disabled. Establishing such interaction (or rather *non*-interaction in this case) between transitions can be exploited to reduce the state space of the transition system as we will see later. Of course, for this purpose, we also need to know which mental state conditions τ_2 affects. To this end, we introduce *write sets* in the next subsection. We compare read sets as treated here with similar notions in the POR literature in Sect. 5.2.3.

5.2.2 Write set

The *write set*, denoted $\mathsf{Write}(\tau, \Psi)$ of a transition τ with respect to a set of mental state conditions Ψ is the subset of Ψ whose truth is changed by at least one step in τ. This is formalised by the

Transition	Read	Write^+	Write^-	Write
τ_0	$\{\psi_0, \psi_9\}$	$\{\psi_5, \psi_6\}$	$\{\psi_0, \psi_9\}$	$\{\psi_0, \psi_5, \psi_6, \psi_9\}$
τ_1	$\{\psi_1, \psi_5\}$	$\emptyset$	$\emptyset$	$\emptyset$
τ_2	$\{\psi_1, \psi_5\}$	$\{\psi_8\}$	$\{\psi_1\}$	$\{\psi_1, \psi_8\}$
τ_3	$\{\psi_3, \psi_8\}$	$\{\psi_4\}$	$\{\psi_8\}$	$\{\psi_4, \psi_8\}$
τ_4	$\{\psi_2, \psi_6\}$	$\emptyset$	$\emptyset$	$\emptyset$
τ_5	$\{\psi_2, \psi_6\}$	$\{\psi_8\}$	$\{\psi_2\}$	$\{\psi_2, \psi_8\}$
τ_6	$\{\psi_3, \psi_8\}$	$\{\psi_4\}$	$\{\psi_8\}$	$\{\psi_4, \psi_8\}$
τ_7	$\{\psi_4, \psi_7\}$	$\emptyset$	$\{\psi_0, \psi_1, \psi_2, \psi_3, \psi_4, \psi_7\}$	$\{\psi_0, \psi_1, \psi_2, \psi_3, \psi_4, \psi_7\}$
τ_8	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

$$
\begin{aligned}
\Psi &= \textstyle\bigcup_{0 \le i \le 8} \mathsf{Read}(\tau_i) \\
&= \{ \quad \psi_0 = \textbf{goal}\texttt{(filled)} \wedge \textbf{bel}\texttt{(not(washed))} \quad , \\
&\qquad \psi_1 = \textbf{goal}\texttt{(filled)} \wedge \textbf{bel}\texttt{(toAdd(bananas))} \quad , \\
&\qquad \psi_2 = \textbf{goal}\texttt{(filled)} \wedge \textbf{bel}\texttt{(toAdd(oranges))} \quad , \\
&\qquad \psi_3 = \textbf{goal}\texttt{(filled)} \wedge \textbf{bel}\texttt{(recipe(F,_))} \quad , \\
&\qquad \psi_4 = \textbf{goal}\texttt{(switch(on))} \wedge \textbf{bel}\texttt{(filled)} \quad , \\
&\qquad \psi_5 = \textbf{bel}\texttt{(washed, added(bananas,Q), Qnew is Q+1)} \quad , \\
&\qquad \psi_6 = \textbf{bel}\texttt{(washed, added(oranges,Q), Qnew is Q+1)} \quad , \\
&\qquad \psi_7 = \textbf{bel}\texttt{(switch(off))} \quad , \\
&\qquad \psi_8 = \textbf{bel}\texttt{(not(ticked(F)), washed, recipe(F,Qr), added(F,Qr))} \quad , \\
&\qquad \psi_9 = \textbf{bel}\texttt{(not(washed))} \quad \}
\end{aligned}
$$

Table 5.3: Read set and write sets with respect to the set Ψ for transitions that occur in the transition system of `blenderAgent`.

following definition.

Definition 9. *Let $\tau \in \Omega_T$ be a transition, and let $\Psi \subseteq \mathcal{L}_\Psi$ be a set of mental state conditions. Then:*

$$\begin{array}{lcl} \mathsf{Write}^+(\tau, \Psi) & = & \bigcup_{\langle \mu, \mu' \rangle \in \tau} \{ \psi \in \Psi \mid \mu \not\models_\Psi \psi \text{ and } \mu' \models_\Psi \psi \} \\ \mathsf{Write}^-(\tau, \Psi) & = & \bigcup_{\langle \mu, \mu' \rangle \in \tau} \{ \psi \in \Psi \mid \mu \models_\Psi \psi \text{ and } \mu' \not\models_\Psi \psi \} \\ \mathsf{Write}(\tau, \Psi) & = & \mathsf{Write}^+(\tau, \Psi) \cup \mathsf{Write}^-(\tau, \Psi) \end{array}$$

We call $\mathsf{Write}^+(\tau, \Psi)$ the *positive write set* with respect to Ψ, and $\mathsf{Write}^-(\tau, \Psi)$ the *negative write set* with respect to Ψ. The set $\mathsf{Write}(\tau, \Psi)$ is sometimes called the *total write set*.

What can we do with write sets? Well, one of their purposes is, as already hinted at when treating read sets, to establish whether a transition can become enabled or disabled when another transition is executed. For example, suppose we have two transitions, say τ_1 and τ_2, and we want to know whether τ_1 can enable τ_2. We know that the read set of τ_2 is $\mathsf{Read}(\tau_2)$. Consequently, if the positive write set of τ_1 with respect to $\mathsf{Read}(\tau_2)$ is non-empty, then there exists a step $\langle \mu, \mu' \rangle \in \tau_1$ and a $\psi \in \mathsf{Read}(\tau_2)$ such that $\mu \not\models_\Psi \psi$ and $\mu' \models_\Psi \psi$. If ψ's falsehood in μ was the only thing prohibiting τ_2 from being enabled in μ, then τ_2 becomes enabled due to execution of τ_1. We define this behaviour of enabling and disabling more formally in Sect. 5.4. Table 5.3 shows the write sets of `blenderAgent`'s transitions with respect to a set Ψ containing all MSCs that are in at least one of these transition's read set. A second use of write sets is to establish whether a transition can affect the truth or falsehood of an LTL formula ϕ. In such case, write sets are computed with respect to the mental state conditions that occur in ϕ. We treat this in more detail in Sect. 5.4.1.

5.2.3 Perspective

In the POR literature, with a few exceptions written exclusively with respect to concurrent systems that are specified in an imperative language (e.g. PROMELA), concepts that are similar to read and write sets exist. For example, the *reference sets* of [96] contain all *system variables* that are read from and written to by a transition. A system variable is the type of variable that one uses when programming in languages like C or JAVA; we use the "system" prefix to distinguish it from the type of variables that are used in Prolog (which are indeed quite different). For example, the transition corresponding to the imperative statement $x := y + z$ reads from 2 variables (i.e. y and z), and writes to 1 variable (i.e. x). In [24], a concept similar to read and write sets exists as well, yet more implicit: when formulating heuristics for the *dependence relation* for transitions (treated in Sect. 5.4), which is also one of the reasons for us to define read and write sets, it is said that "pairs of transitions that share a variable, which is changed by at least one of them, are dependent".

One may regard the way we treat mental state conditions as if an MSC is a boolean system variables in an imperative language that transitions can write to (set them to true or false), and read from. The terms "read set" and "write set" are in fact derived from Ex. 2.1 in [46] in which boolean system variables occur that can be "read from" and "written to". There is, however, a significant difference between GOAL and imperative languages in this respect: the write set of a transition in an imperative language can be determined from straightforward static inspection of the code, because the respective variables are always mentioned explicitly in the statements that write to them (x in the example above). Such static analyses are used by POR algorithms to determine *before* generation of the entire transition system how a transition τ_1 can enable or disable a transition τ_2. For GOAL, in contrast, such static analyses are complicated by the following reasons:

1. MSCs become true and false due to additions to and deletions from the belief and goal bases that a transition brings about. However:
   a) MSCs that are written to (i.e. change truth value) do not occur explicitly in the "statements" (i.e. its action rules and actions) of the GOAL agent's code. Instead, only the

additions to and deletions from the belief and goal bases, which may cause the truth or falsehood of MSCs to change, can be determined by inspection of the agent's code.

b) Not *all* additions and deletions that a transition brings about occur in the agent's code. More specifically:

   i. If a goal is achieved after execution of a transition, then this goal is removed from the goal base. Such goals do not occur in the agent's code, however.

   ii. In the code, free PL variables that are instantiated at runtime may occur. From static code analysis, the specific instantiations cannot be determined.

c) The additions and deletions that a transition brings about may be the *indirect* cause of the truth value of an MSC ψ to change. That is, if ψ contains predicates that are defined by PL rules in the knowledge base, then the additions or deletions may cause the body of these rules to become derivable or underivable such that the truth value of ψ changes. To detect this, we need to reason about PL rules without knowing what the rest of the mental state looks like: with static code analysis, we can only determine the contents of the initial mental state, but not of all the others.

2. Whether MSCs are written to depends on the contents of the mental state before execution of a transition. This is the reason why in the definition of write sets, the $\bigcup$ operator is applied to all steps belonging to a transition.

Note that such considerations are irrelevant if the entire transition system has already been generated. In such case, we can straightforwardly iterate over all the steps belonging to a transition and compare belief and goal bases before and after execution. However, POR and PBS algorithms try to avoid the generation of the entire transition system such that this straightforward approach cannot be applied in practice.

More specifically, POR and PBS algorithms compute write sets of transitions *before* the transition system of a GOAL agent is generated. The only information that we have at that time is the code by which the agent is defined. Hence, the implemented algorithms for write set computation are based on analysis of this code exclusively (like existing algorithms). However, due to the aforementioned complicating factors, these computations yield approximations rather than precise write sets. Although this does not impair correctness of POR and PBS algorithms, the reductions that they yield are usually better when computations are more precise. In the next section, we discuss our approximation algorithm for write sets.

5.3 Write Set Computation

In this section, we discuss how write sets can be computed based on static analysis of the code by which a GOAL agent is defined. We first give an informal account in Sect. 5.3.1. Subsequently, in Sect. 5.3.2, we give formal definitions that can be used to compute approximate write sets. The proof of correctness can be found in Appx. C.1. We remark that algorithms as presented here are not found in the literature, and are very GOAL specific. The reason is that write set computation is much more complex for GOAL than for programs written in imperative languages as discussed in Sect. 5.2.3.

5.3.1 Approximate Write Sets: Informal Account

Let us first enumerate what we exactly aim at. We want to:

1. compute approximate write sets that are as precise as possible,

2. before the transition system is generated for all $\tau \in \Omega_T$.

We first have a closer look at Item 1. Let Ψ be a set of mental state conditions, and let the approximate write sets for a transition τ with respect to Ψ be denoted by $\mathsf{ApproxWrite}^{+}(\tau, \Psi)$,

$\mathsf{ApproxWrite}^-(\tau,\Psi)$, and $\mathsf{ApproxWrite}(\tau,\Psi)$. What does it mean for these sets to "approximate" the precise write sets of τ? Well, here, "approximation" means *over-approximation*, e.g. *at least* all the mental state conditions that occur in the precise positive write set $\mathsf{Write}^+(\tau,\Psi)$ must also occur in $\mathsf{ApproxWrite}^+(\tau,\Psi)$. More formally, we require $\mathsf{Write}^+(\tau,\Psi) \subseteq \mathsf{ApproxWrite}^+(\tau,\Psi)$. Similar conditions can be formulated for the negative and total write sets, i.e. $\mathsf{Write}^-(\tau,\Psi) \subseteq \mathsf{ApproxWrite}^-(\tau,\Psi)$ and $\mathsf{Write}(\tau,\Psi) \subseteq \mathsf{ApproxWrite}(\tau,\Psi)$. We remark that $\mathsf{ApproxWrite}^+(\tau,\Psi)$ and $\mathsf{ApproxWrite}^-(\tau,\Psi)$ are not each others dual: they are *both* over-approximations, which means that a $\psi \in \Psi$ may be both in $\mathsf{ApproxWrite}^+(\tau,\Psi)$ and $\mathsf{ApproxWrite}^-(\tau,\Psi)$. For example, if a transition adds `foo(bar)` to the belief base and deletes `foo(baz)` from it, then the mental state condition **bel**`(foo(X))` can become both true and false, depending on the instantiation of `X` (this instantiation is not known at the time approximate write sets are computed).

As aforementioned, the use of approximate (instead of precise) write sets with Por and Pbs algorithms does not impair their correctness (see also Sect. 5.4). However, the performance of these algorithms in terms of the obtained reduction of the transition system is better when computations are more precise; the definitions that we present in this section are the result of a process of refining and improving an initial set of simple yet coarse definitions.

In the remainder of this subsection, we focus on Item 2. Because computation of approximate write sets should occur before the transition system is generated, the algorithm for doing this can only make use of the code by which the Goal agent is defined. More specifically, it uses information contained by the knowledge base (defined by the **knowledge** section of the code), the action rules (defined by the **program** section of the code), and the specifications of user-defined actions (defined by the **actionspec** section of the code); the initial mental state is *not* taken into consideration. This is because write sets need to be computable without reference to mental states. Under this constraint, the best we can do is compute what *might* happen using approximations.

The algorithm works by computing for each action rule ρ an approximate write set such that this set is an approximate write set for *all* transitions that can be generated by ρ. Because every transition can be generated by at least one action rule (except the **skip** transition, which trivially has an empty write set always), all $\tau \in \Omega_T$ are covered as required. We call the set of transitions that can be generated by ρ the *transition class* of ρ, and denote it by $\boldsymbol{\tau}(\rho)$. Because approximate write sets are the same for all transitions belonging to $\boldsymbol{\tau}(\rho)$, in the remainder, we denote approximate write sets by $\mathsf{ApproxWrite}^+(\boldsymbol{\tau}(\rho),\Psi)$, $\mathsf{ApproxWrite}^-(\boldsymbol{\tau}(\rho),\Psi)$, and $\mathsf{ApproxWrite}(\boldsymbol{\tau}(\rho),\Psi)$.

Let $\boldsymbol{\tau}(\rho)$ be a transition class in which $\rho =$ **if** ψ **then** α. Approximate write sets with respect to Ψ are then computed as follows.

1. First, an approximation of the change that the transitions in $\boldsymbol{\tau}(\rho)$ can bring about is computed. This computation is based on information contained in α, and represented by the *approximate add/del function* $\mathsf{Approx}\Delta$. The result of applying $\mathsf{Approx}\Delta$ to $\boldsymbol{\tau}(\rho)$ is similar to the result of Δ: $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$. Informally, $\mathsf{ApproxAdd}_\Sigma$ contains all *belief templates*, i.e. uninstantiated facts, whose instantiations may be added to the belief base by execution of a transition in $\boldsymbol{\tau}(\rho)$. Similarly, $\mathsf{ApproxDel}_\Sigma$ contains all belief templates whose instantiations may be deleted from the belief base by execution of a transition in $\boldsymbol{\tau}(\rho)$. Likewise, $\mathsf{ApproxAdd}_\Gamma$ and $\mathsf{ApproxDel}_\Gamma$ contain all *goal templates* whose instantiations may, respectively, be added to or deleted from the goal base by execution of a transition belonging to $\boldsymbol{\tau}(\rho)$. We now have a closer look at how these sets can be computed.

   a) $\mathsf{ApproxAdd}_\Sigma$ can only be non-empty if α is a user-defined action (**adopt** and **drop** actions cannot cause beliefs to be added). In such case, all positive literals in α's postcondition are added to $\mathsf{ApproxAdd}_\Sigma$. For example, if $\alpha =$ **insert**`(foo(X))` then $\mathsf{ApproxAdd}_\Sigma = \{$`foo(X)`$\}$.

   b) $\mathsf{ApproxDel}_\Sigma$ is computed similar to $\mathsf{ApproxAdd}_\Sigma$: it can only be non-empty if α is a user-defined action (**adopt** and **drop** actions cannot cause beliefs to be deleted) in which case all negative literals in α's postconditions are added to $\mathsf{ApproxDel}_\Sigma$. For example, if $\alpha =$ **delete**`(foo(X))` then $\mathsf{ApproxDel}_\Sigma = \{$`foo(X)`$\}$.

c) $\mathsf{ApproxAdd}_\Gamma$ can only be non-empty if α is an **adopt** action (user-defined and **drop** actions cannot cause goals to be added). In such case, the goal template contained by α's argument is added to $\mathsf{ApproxAdd}_\Gamma$. For example, if $\alpha =$ **adopt**`(foo(X),bar(Y))`, then $\mathsf{ApproxAdd}_\Gamma = \{$`foo(X), bar(Y)`$\}$.

d) Whereas the previous three sets can be computed quite straightforwardly, $\mathsf{ApproxDel}_\Gamma$'s computation is more involved. This works as follows.

i. First, a set containing all goal templates that occur in the agent's code is constructed. This set is denoted by *Goals* and includes the goals in the initial goal base, as well as all the goal templates occurring in **adopt** actions. For example, suppose the initial goal base only contains the goal `foo(bar)`, while the following **adopt** actions occur: **adopt**`(foo(X))` and **adopt**`(baz(qux,quux),corge(X))`. Then: *Goals* $=\{$`foo(bar)`, `foo(X)`, `baz(qux,quux), corge(X)`$\}$ (note the difference between "`,`" and ","; recall from Sect. 2.2 that the former denotes PL's conjunction operator, while the latter is used as a separator for elements in a set).

ii. Second, the action α is analysed. Both user-defined and **drop** actions can cause a goal to be deleted from the goal base: in the former case, the goal is achieved due to changes brought about to the belief base, whereas in the latter case, the argument of the **drop** action is derivable from the goal.

A. If α is a user-defined action, then the algorithm compares each goal template in *Goals* to the changes that α might bring about to the belief base, which are already computed as $\mathsf{ApproxAdd}_\Sigma$ and $\mathsf{ApproxDel}_\Sigma$. Informally, if a goal template in *Goals* contains a fact that *accords with* the change brought about to the belief base, then this template is added to $\mathsf{ApproxDel}_\Gamma$. For example, if $\mathsf{ApproxAdd}_\Sigma$ and *Goals* are defined as in the examples above, then $\mathsf{ApproxDel}_\Gamma = \{$`foo(bar)`, `foo(X)`$\}$, i.e. both `foo(bar)` and `foo(X)` are said to accord with `foo(X)`, but `baz(qux,quux)` and `corge(X)` do not. At this point, we are deliberately a bit vague about when two facts accord with each other as this is too technical for our current exposition; here, we only aim at providing a rough outline of the computation.

B. If α is a **drop** action, then the algorithm investigates each goal template in *Goals* in a quite similar fashion as for the user-defined action case. For example, if $\alpha =$ **drop**`(baz(qux,Y))` and *Goals* is defined as before, then $\mathsf{ApproxDel}_\Gamma = \{$`baz(qux,quux), corge(X)`$\}$, i.e. `baz(qux,Y)` is said to accord with `baz(qux,quux)`, and as such the entire goal template in which `baz(qux,quux)` occurs is added to $\mathsf{ApproxDel}_\Gamma$. Again, we defer the details to the next subsection.

2. Computation of the approximate add/del function for $\boldsymbol{\tau}(\rho)$ is yet insufficient for determining whether the truth value of mental state conditions in Ψ can be changed by execution of a transition belonging to $\boldsymbol{\tau}(\rho)$. The reason is that the *explicit* change that $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho))$ describes might cause facts to become derivable or underivable from the knowledge base when a transition τ belonging to $\boldsymbol{\tau}(\rho)$ is executed; these derivable facts are of relevance when predicates that are defined by rules in the knowledge base occur in MSCs in Ψ.

   We outline the procedure for determining whether a fact χ becomes derivable from the belief and knowledge bases combined when τ is executed as follows. Suppose χ can only be derived using rules in the knowledge base. Then, there exists a rule, say r, whose head is unifiable with χ, and whose body is derivable after execution of τ, but not before (otherwise, χ is already derivable before τ's execution, and did not *become* so). Suppose, for simplicity, that r's body is a conjunction of facts. Then, there occurs at least one fact in this conjunction, say χ', that becomes derivable by execution of τ. This fact is either derivable from another rule in the knowledge base or from a fact that is added to the belief base. In the former case, the previous discussion is inductively applied. In the latter case, there exists a belief template in the previously computed $\mathsf{ApproxAdd}_\Sigma$ that accords with χ'.

We illustrate the previous with the following example. Let the knowledge base $K = \{r\}$ in which $r =$ `foo :- bar,baz(qux,Y)`, let $\alpha =$ **insert**`(baz(X,quux))` such that $\mathsf{ApproxAdd}_\Sigma = \{$`baz(X,quux)`$\}$, and let $\chi =$ `foo`. Obviously, r's head is unifiable with χ such that the body of this rule is investigated. For the first conjunct, `bar`, there is no fact in $\mathsf{ApproxAdd}_\Sigma$ that accords with it. For the second conjunct, `baz(qux,Y)` there is: `baz(X,quux)`. Consequently, χ may have become derivable by execution of transitions in the transition class $\boldsymbol{\tau}(\rho)$ in which $\rho =$ **if** ψ **then insert**`(baz(X,quux))`.

The previous example also shows that our algorithm over-approximates the facts that really become derivable from executing transitions in $\boldsymbol{\tau}(\rho)$. For one thing, by only requiring a single conjunct to become derivable, we tacitly assume that all other conjuncts are already derivable (`bar` in the example above). This need of course not be the case, but it is in line with the fact that we cannot inspect mental states during the static code analysis. Also, because $\mathsf{ApproxAdd}_\Sigma$ is an approximation, it might not be that the templates it contains are instantiated at runtime in such a way that they really make a body's conjunct derivable. In the example, this happens if `X` is never bound to `qux`.

We remark that in the next subsection, we drop the assumption that rule bodies must be conjunctions of facts; this constraint was adopted here for the sake of the exposition. Also, we remark that the incorporation of knowledge is also necessary when we want to compute Del_Γ. This is further detailed in the next subsection.

3. Given the above two steps, the remainder is quite straightforward. We can iterate over all mental state conditions in Ψ, and determine for each such Msc whether there occurs a mental literal in it whose truth value can change, based on $\mathsf{Approx}\Delta$ combined with knowledge. If this is the case, the respective mental state condition is added to the approximate write set. How this works exactly is actually easier explained in the more formal discussion with which we now proceed.

5.3.2 Approximate Write Sets: Formal Account

Let us start with a formal definition of transition classes. Recall that the transition class $\boldsymbol{\tau}(\rho)$ of an action rule ρ is the set of transitions that can be generated by ρ.

Definition 10. *Let ρ be an action rule, and let Ω_T be the set of all transitions. Then:*

$$\boldsymbol{\tau}(\rho) \quad = \quad \{\tau \in \Omega_T \mid \rho = \mathsf{Rule}(\tau)\}$$

The set of all transition classes is denoted by $\boldsymbol{\Omega_T}$.

We exemplify the concept of transition classes with `blenderAgent`. Its **program** section defines 5 action rules (see Fig. 2.1 in Chap. 2):

ρ_0 = **if goal**`(filled)` $\wedge$ **bel**`(not(washed))` **then** `wash`
ρ_1 = **if goal**`(filled)` $\wedge$ **bel**`(toAdd(bananas))` **then** `add(bananas,1)`
ρ_2 = **if goal**`(filled)` $\wedge$ **bel**`(toAdd(oranges))` **then** `add(oranges,1)`
ρ_3 = **if goal**`(filled)` $\wedge$ **bel**`(recipe(F,_))` **then** `tick(F)`
ρ_4 = **if goal**`(switch(on))` $\wedge$ **bel**`(filled)` **then** `blend`

That is, $R = \{\rho_0, \rho_1, \rho_2, \rho_3, \rho_4\}$. Combined with Table 5.2, it follows that `blenderAgent` has 5 transition classes, namely $\boldsymbol{\tau}(\rho_0) = \{\tau_0\}$, $\boldsymbol{\tau}(\rho_1) = \{\tau_1, \tau_2\}$, $\boldsymbol{\tau}(\rho_2) = \{\tau_4, \tau_5\}$, $\boldsymbol{\tau}(\rho_3) = \{\tau_3, \tau_6\}$, and $\boldsymbol{\tau}(\rho_4) = \{\tau_7\}$. Note that τ_8 is not a member of any transition class, because it is generated by the **skip** action, which is only an option if no action rules are applicable.

We proceed our exposition according to the same structure by which we introduced the algorithm informally in the previous subsection. We present our algorithm as a set of definitions that can be computed straightforwardly to obtain approximate write sets. After introducing such a definition, we point to the relevant propositions, lemmas, or theorems in Appx. C.1, which are

used to prove that write sets are over-approximated correctly. In the following, let Ψ be a set of mental state conditions, let $\rho = \textbf{if } \psi \textbf{ then } \alpha$ be an action rule, and let $\boldsymbol{\tau}(\rho)$ be the corresponding transition class.

1. In this step, the approximate add/del function $\mathsf{Approx}\Delta$ is computed for $\boldsymbol{\tau}(\rho)$, i.e. the four sets occurring in $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$.

   a) In this step, the set $\mathsf{ApproxAdd}_\Sigma$ is computed. Recall that $\mathsf{ApproxAdd}_\Sigma$ contains all belief templates whose instantiations may be added to the belief base by execution of a transition in $\boldsymbol{\tau}(\rho)$. Also, recall from Sect. 2.3.2 that $\mathit{literals}^+(\chi)$ is the set of positive literals in χ (provided that χ is a conjunction of literals). Then, $\mathsf{ApproxAdd}_\Sigma$ is defined as follows.

   Definition 11. *Let $\boldsymbol{\tau}(\rho)$ be a transition class in which $\rho = \textbf{if } \psi \textbf{ then } \alpha$ such that $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$.*
   Then:

   $$\mathsf{ApproxAdd}_\Sigma \quad = \quad \begin{cases} \mathit{literals}^+(\chi_{\mathtt{post}}) & \text{if } \alpha = \langle \chi_{\mathtt{pre}}, \chi_{\mathtt{post}} \rangle \\ \emptyset & \text{otherwise} \end{cases}$$

   Proposition 1 in Appx. C.1.1 states that all facts that are derivable from the set of facts that are added to the belief base due to the execution of a transition in $\boldsymbol{\tau}(\rho)$ are also derivable from $\mathsf{ApproxAdd}_\Sigma$.

   b) In this step, the set $\mathsf{ApproxDel}_\Sigma$ is computed. Recall that $\mathsf{ApproxDel}_\Sigma$ contains all belief templates whose instantiations may be deleted from the belief base by execution of a transition in $\boldsymbol{\tau}(\rho)$. Also, recall from Sect. 2.3.2 that $\mathit{literals}^-(\chi)$ is the set of negative literals in χ (provided that χ is a conjunction of literals). Then, $\mathsf{ApproxDel}_\Sigma$ is defined as follows.

   Definition 12. *Let $\boldsymbol{\tau}(\rho)$ be a transition class in which $\rho = \textbf{if } \psi \textbf{ then } \alpha$ such that $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$.*
   Then:

   $$\mathsf{ApproxDel}_\Sigma \quad = \quad \begin{cases} \{\chi \mid \mathtt{not}(\chi) \in \mathit{literals}^-(\chi_{\mathtt{post}})\} & \text{if } \alpha = \langle \chi_{\mathtt{pre}}, \chi_{\mathtt{post}} \rangle \\ \emptyset & \text{otherwise} \end{cases}$$

   Proposition 2 in Appx. C.1.1 states that all facts that are derivable from the set of facts that are deleted from the belief base due to the execution of a transition in $\boldsymbol{\tau}(\rho)$ are also derivable from $\mathsf{ApproxDel}_\Sigma$.

   c) In this step, the set $\mathsf{ApproxAdd}_\Gamma$ is computed. Recall that $\mathsf{ApproxAdd}_\Gamma$ contains all goal templates whose instantiations may be added to the goal base by execution of a transition in $\boldsymbol{\tau}(\rho)$. Then, $\mathsf{ApproxAdd}_\Gamma$ is defined as follows.

   Definition 13. *Let $\boldsymbol{\tau}(\rho)$ be a transition class in which $\rho = \textbf{if } \psi \textbf{ then } \alpha$ such that $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$.*
   Then:

   $$\mathsf{ApproxAdd}_\Gamma \quad = \quad \begin{cases} \{\chi\} & \text{if } \alpha = \textbf{adopt}(\chi) \\ \emptyset & \text{otherwise} \end{cases}$$

   Proposition 3 in Appx. C.1.1 states that all facts that are derivable from a goal that is added due to the execution of a transition belonging to $\boldsymbol{\tau}(\rho)$ are also derivable from a goal in $\mathsf{ApproxAdd}_\Gamma$.

   d) In this step, the set $\mathsf{ApproxDel}_\Gamma$ is computed based on *Goals*. Recall that $\mathsf{ApproxAdd}_\Gamma$ contains all goal templates whose instantiations may be deleted from the goal base by execution of a transition in $\boldsymbol{\tau}(\rho)$.

      i. In this step, the set *Goals* is computed. Recall that *Goals* contains all goal templates that occur in the agent's code.

Definition 14. *Let Γ_0 be the initial goal base, and let R be the set of action rules. Then:*

$$Goals \quad = \quad \Gamma_0 \cup \{\chi \mid \textbf{if } \psi \textbf{ then adopt}(\chi) \in R\}$$

ii. In this step, $\mathsf{ApproxDel}_\Gamma$ is computed depending on whether α is a user-defined action or a **drop** action.

Definition 15. *Let $\boldsymbol{\tau}(\rho)$ be a transition class in which $\rho =$ **if** ψ **then** α such that $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$. Then:*

$$\mathsf{ApproxDel}_\Gamma \quad = \quad \begin{cases} \{\gamma \in Goals \mid \text{there exists a } \chi \in \gamma \text{ such that} \\ \qquad der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)\} & \\ \qquad\qquad \text{if } \alpha = \langle \chi_{\mathtt{pre}}, \chi_{\mathtt{post}}\rangle & \\ \{\gamma \in Goals \mid der^+(K, \gamma, \{\mathtt{true}\}, \chi)\} & \text{if } \alpha = \textbf{drop}(\chi) \\ \emptyset & \text{otherwise} \end{cases}$$

Note the occurrence of the der^+ relation in the above definition. This relation is used to incorporate knowledge, and treated in the next step. In the next step, we also elaborate on der^+'s use in the above definition.

Proposition 4 in Appx. C.1.1 states that all facts that are derivable from a goal that is deleted due to the execution of a transition belonging to $\boldsymbol{\tau}(\rho)$ are also derivable from a goal in $\mathsf{ApproxDel}_\Gamma$.

2. In this step, based on the computation of $\mathsf{Approx}\Delta$, we incorporate knowledge under the following constraint:

Constraint 1. *The only built-in* Pl *predicates that occur in a* Goal *program and property specification are:* `not/1`, `,/2`, `=/2`, *and all arithmetic predicates and functions, e.g.* `>/2`, `</2`, `=</2`, `>=/2`, `+/2`, *and* `-/2`.

We discuss why we impose this constraint below the next definition. To incorporate knowledge, we introduce two relations, denoted $der^+ \subseteq 2^{\mathcal{L}}\times 2^{\mathcal{L}}\times 2^{\mathcal{L}}\times\mathcal{L}$ and $der^- \subseteq 2^{\mathcal{L}}\times 2^{\mathcal{L}}\times 2^{\mathcal{L}}\times\mathcal{L}$, that express how a term $\chi \in \mathcal{L}$ (fourth argument) according to Constr. 1 can become derivable or underivable when a set of facts $X^+ \in 2^{\mathcal{L}}$ (second argument) is added to, and a set of facts $X^- \in 2^{\mathcal{L}}$ (third argument) is deleted from a set of facts X combined with a set of rules $X^K \in 2^{\mathcal{L}}$ (first argument).

Definition 16. *Let X^K be a set of rules, let X^+ and X^- be sets of facts, and let $\chi \in \mathcal{L}$ be a term according to Constr. 1. Then:*

$der^+(X^K, X^+, X^-, \chi)$ iff χ is a fact and:
- $X^+ \models \chi$, or
- there exists a χ_h `:-` $\chi_b \in X^K$ such that
  there exists a substitution θ such that
  $\chi_h\theta = \chi\theta$ and $der^+(X^K, X^+, X^-, \chi_b\theta)$

$der^+(X^K, X^+, X^-, \mathtt{not}(\chi))$ iff $\mathtt{true} \in X^-$ or $der^-(X^K, X^+, X^-, \chi)$

$der^+(X^K, X^+, X^-, \chi_1\mathtt{,}\chi_2)$ iff $der^+(X^K, X^+, X^-, \chi_1)$ or $der^+(X^K, X^+, X^-, \chi_2)$

Without Constr. 1, we should have defined der^+ (and der^- below) for all built-in Pl predicates, which would not have yielded any new fundamental insights. Note that we have not defined der^+ for the unification and arithmetic predicates and functions. The reason that der^+ is always false for such terms is because they cannot *become* (un)derivable in isolation: they are always derivable (e.g. `2<3`), always underivable (e.g. `2<1`), or can become

(un)derivable because a variable is bound (e.g. `2<X` and `X` can become bound to `3` when previously it could not), in which case there must exist another term that becomes derivable in such a way that it binds the variable accordingly.

The appearance of "or" in Def. 16 in the $\chi_1\texttt{,}\chi_2$ may seem a bit odd: one might expect "and" instead. The reason for the disjunctive definition, however, is that for $\chi_1\texttt{,}\chi_2$ to become derivable, *at least* χ_1 *or* χ_2 needs to become derivable. Consequently, if either of the two conjuncts becomes derivable, the conjunction may become derivable, which is what the use of "or" expresses. We will see something similar in the definition of der^- below.

We have already encountered two typical uses of this relation, namely in the definition of $\mathsf{ApproxDel}_\Gamma$ in Def. 12. The first use is with $X^K = K$ and $X^+ = \mathsf{ApproxAdd}_\Sigma$ and $X^- = \mathsf{ApproxDel}_\Sigma$ and χ is a goal conjunct of a goal in *Goals*. In such case, we compute der^+ to determine whether this goal conjunct becomes derivable from adding $\mathsf{ApproxAdd}_\Sigma$ to a set X, and deleting $\mathsf{ApproxDel}_\Sigma$ from X; this set of facts X represents a belief base.

The second use is with $X^K = K$ and X^+ is a goal template from *Goals* and $X^- = \{\texttt{true}\}$ and χ is the argument of a **drop** action. In such case, we compute der^+ to determine whether this argument is derivable from a goal template in *Goals*. One might deem the occurrence of `true` in X^- a bit odd, as we described X^- earlier as a set of facts that are deleted: how can `true` ever be deleted? Well, it cannot: we treat the occurrence of `true` in X^- as a *wild card* that is used in the definition of der^+ in case of $\texttt{not}(\chi)$. The reason is that because goals in *Goals* are uninstantiated templates, we can over-approximate which facts are derivable but can only under-approximate which facts are underivable. In case `not` occurs in the **drop** action's argument, these underivable facts play a role, but we have insufficient information to determine these facts precise enough. For correctness of the algorithm to be preserved, the only thing we can do in such cases is assume the der^+ relation to be true. This is used in the proofs of Lemmas 3,4 in Appx. C.1.2 (see below for a brief introduction of these lemmas).

The definition of der^- is similar to that of der^+.

Definition 17. *Let X^K be a set of rules, let X^+ and X^- be sets of facts, and let $\chi \in \mathcal{L}$ be a term according to Constr. 1.*
Then:

$der^-(X^K, X^+, X^-, \chi)$ iff χ is a fact and:
- $X^- \models \chi$, or
- there exists a $\chi_h \texttt{ :- } \chi_b \in X^K$ such that there exists a substitution θ such that $\chi_h\theta = \chi\theta$ and $der^-(X^K, X^+, X^-, \chi_b\theta)$

$der^-(X^K, X^+, X^-, \texttt{not}(\chi))$ iff $\texttt{false} \in X^+$ or $der^+(X^K, X^+, X^-, \chi)$

$der^-(X^K, X^+, X^-, \chi_1\texttt{,}\chi_2)$ iff $der^-(X^K, X^+, X^-, \chi_1)$ or $der^-(X^K, X^+, X^-, \chi_2)$

In Appx. C.1.2, four lemmas (i.e. Lemmas 1,2,3,4) are introduced and proven that concern der^+. Let $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle\mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle\mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$, let K be the knowledge base, and let χ be a term according to Constr. 1. Lemma 1 states that if the execution of a transition belonging to $\boldsymbol{\tau}(\rho)$ can make χ derivable from the belief and knowledge bases, then $der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma, \chi)$ is true. Likewise, Lemma 2 states that if the execution of a transition belonging to $\boldsymbol{\tau}(\rho)$ can make χ underivable from the belief and knowledge bases, then $der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma, \chi)$ is true. Lemma 3 states that if χ is derivable from a goal (combined with knowledge) that is added to the goal base due to the execution of a transition in $\boldsymbol{\tau}(\rho)$, then there exists a $\gamma \in \mathsf{ApproxAdd}_\Gamma$ such that $der^+(K, \gamma, \{\texttt{true}\}, \chi)$ is true. Finally, Lemma 4 states that if χ is derivable from a goal (combined with knowledge) that is deleted from the goal base due to the execution of a transition in $\boldsymbol{\tau}(\rho)$, then there exists a $\gamma \in \mathsf{ApproxDel}_\Gamma$ such that $der^+(K, \gamma, \{\texttt{true}\}, \chi)$.

3. In this final step, the approximate write sets are computed. First, in the same spirit of der^+ and der^-, we introduce the relations $Der^+ \subseteq \mathbf{\Omega_T} \times \mathcal{L}_\Psi$ and $Der^- \subseteq \mathbf{\Omega_T} \times \mathcal{L}_\Psi$ (recall that $\mathbf{\Omega_T}$ is the set of all transition classes). Informally, $Der^+(\boldsymbol{\tau}(\rho), \psi)$ expresses that ψ can become true if the changes described by $\mathsf{ApproxΔ}(\boldsymbol{\tau}(\rho))$ are brought about; $Der^-(\boldsymbol{\tau}(\rho), \psi)$ expresses the converse, i.e. ψ can become false if the changes described by $\mathsf{ApproxΔ}(\boldsymbol{\tau}(\rho))$ are brought about. The formal definition occurs below; recall from Sect. 2.3.1 that for notational convenience, we regard a conjunction of facts (i.e. a goal) as a set containing all conjuncts (and nothing else).

Definition 18. *Let K be the knowledge base, and let $\boldsymbol{\tau}(\rho) \in \mathbf{\Omega_T}$ be a transition class such that $\mathsf{ApproxΔ}(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle\rangle$.*
Then:

$$
\begin{array}{lll}
Der^+(\boldsymbol{\tau}(\rho), \mathbf{bel}(\chi)) & \text{iff} & der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi) \\
Der^+(\boldsymbol{\tau}(\rho), \mathbf{goal}(\chi)) & \text{iff} & \text{there exists a } \gamma \in \mathsf{ApproxAdd}_\Gamma \text{ s.t. } der^+(K, \gamma, \{\mathtt{true}\}, \chi) \\
Der^+(\boldsymbol{\tau}(\rho), \neg\psi) & \text{iff} & Der^-(\boldsymbol{\tau}(\rho), \psi) \\
Der^+(\boldsymbol{\tau}(\rho), \psi_1 \wedge \psi_2) & \text{iff} & Der^+(\boldsymbol{\tau}(\rho), \psi_1) \text{ or } Der^+(\boldsymbol{\tau}(\rho), \psi_2)
\end{array}
$$

Definition 19. *Let K be the knowledge base, and let $\boldsymbol{\tau}(\rho) \in \mathbf{\Omega_T}$ be a transition class such that $\mathsf{ApproxΔ}(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle\rangle$.*
Then:

$$
\begin{array}{lll}
Der^-(\boldsymbol{\tau}(\rho), \mathbf{bel}(\chi)) & \text{iff} & der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi) \\
Der^-(\boldsymbol{\tau}(\rho), \mathbf{goal}(\chi)) & \text{iff} & \text{there exists a } \gamma \in \mathsf{ApproxDel}_\Gamma \text{ s.t. } der^+(K, \gamma, \{\mathtt{true}\}, \chi) \\
Der^-(\boldsymbol{\tau}(\rho), \neg\psi) & \text{iff} & Der^+(\boldsymbol{\tau}(\rho), \psi) \\
Der^-(\boldsymbol{\tau}(\rho), \psi_1 \wedge \psi_2) & \text{iff} & Der^-(\boldsymbol{\tau}(\rho), \psi_1) \text{ or } Der^-(\boldsymbol{\tau}(\rho), \psi_2)
\end{array}
$$

Lemmas 5,6 in Appx. C.1.3 establish that if a mental state condition becomes true (or false) by execution of a transition belonging to $\boldsymbol{\tau}(\rho)$, then $Der^+(\boldsymbol{\tau}(\rho), \psi)$ is true (or $Der^-(\boldsymbol{\tau}(\rho), \psi)$ is true).

Given the definitions of Der^+ and Der^-, computation of the approximate write set with respect to Ψ straightforward: for each $\psi \in \Psi$, it is determined whether Der^+ and Der^- are true provided $\mathsf{ApproxAdd}_\Sigma$, $\mathsf{ApproxAdd}_\Gamma$, $\mathsf{ApproxDel}_\Sigma$, and $\mathsf{ApproxDel}_\Gamma$.

Definition 20. *Let $\boldsymbol{\tau}(\rho) \in \mathbf{\Omega_T}$ be a transition class, and let $\Psi \in \mathcal{L}_\Psi$ be a set of mental state conditions. Then:*

$$
\begin{array}{lcl}
\mathsf{ApproxWrite}^+(\boldsymbol{\tau}(\rho), \Psi) & = & \{\psi \in \Psi \mid Der^+(\boldsymbol{\tau}(\rho), \psi)\} \\
\mathsf{ApproxWrite}^-(\boldsymbol{\tau}(\rho), \Psi) & = & \{\psi \in \Psi \mid Der^-(\boldsymbol{\tau}(\rho), \psi)\} \\
\mathsf{ApproxWrite}(\boldsymbol{\tau}(\rho), \Psi) & = & \mathsf{ApproxWrite}^+(\boldsymbol{\tau}(\rho), \Psi) \cup \mathsf{ApproxWrite}^-(\boldsymbol{\tau}(\rho), \Psi)
\end{array}
$$

Given the propositions and lemmas in Appx. C.1, we can prove that the above definitions of approximate write sets $\mathsf{ApproxWrite}^+(\boldsymbol{\tau}(\rho), \Psi)$, $\mathsf{ApproxWrite}^-(\boldsymbol{\tau}(\rho), \Psi)$ and $\mathsf{ApproxWrite}(\boldsymbol{\tau}(\rho), \Psi)$ indeed yield proper over-approximations of the precise write sets of all transitions belonging to $\boldsymbol{\tau}(\rho)$. We do this in the following three theorems. Theorem 1 states that an approximate positive write set compute for a transition class $\boldsymbol{\tau}(\rho)$ is a proper over-approximation of all precise positive write sets for transitions belonging to $\boldsymbol{\tau}(\rho)$. Theorem 2 states the same for negative write sets (recall from Sect. 5.3.1 that $\mathsf{ApproxWrite}^+(\tau, \Psi)$ and $\mathsf{ApproxWrite}^-(\tau, \Psi)$ are are both over-approximations). Finally, Theorem 3 combines these results for total write sets.

Theorem 1. *Let $\boldsymbol{\tau}(\rho)$ be a transition class, and let $\tau \in \boldsymbol{\tau}(\rho)$. Then:*

$$
\mathsf{Write}^+(\tau, \Psi) \quad \subseteq \quad \mathsf{ApproxWrite}^+(\boldsymbol{\tau}(\rho), \Psi)
$$

Proof. Suppose the theorem is false, i.e. there exists a $\psi \in \Psi$ such that $\psi \in \mathsf{Write}^+(\tau, \Psi)$ and $\psi \notin \mathsf{ApproxWrite}^+(\boldsymbol{\tau}(\rho), \Psi)$.
Then, by definition of Write^+ in Def. 9, there exists a $\langle \mu, \mu' \rangle \in \tau$ such that $\mu \not\models_\Psi \psi$ and $\mu' \models_\Psi \psi$.
Then, by Lemma. 5, $Der^+(\boldsymbol{\tau}(\rho), \psi)$.
Then, by definition of $\mathsf{ApproxWrite}^+$ in Def. 20, $\psi \in \mathsf{ApproxWrite}^+(\boldsymbol{\tau}(\rho), \Psi)$, which yields a contradiction.
This establishes the theorem. □

Theorem 2. *Let $\boldsymbol{\tau}(\rho)$ be a transition class, and let $\tau \in \boldsymbol{\tau}(\rho)$. Then:*

$$\mathsf{Write}^-(\tau, \Psi) \quad \subseteq \quad \mathsf{ApproxWrite}^-(\boldsymbol{\tau}(\rho), \Psi)$$

Proof. Suppose the theorem is false, i.e. there exists a $\psi \in \Psi$ such that $\psi \in \mathsf{Write}^-(\tau, \Psi)$ and $\psi \notin \mathsf{ApproxWrite}^-(\boldsymbol{\tau}(\rho), \Psi)$.
Then, by definition of Write^- in Def. 9, there exists a $\langle \mu, \mu' \rangle \in \tau$ such that $\mu \models_\Psi \psi$ and $\mu' \not\models_\Psi \psi$.
Then, by Lemma. 6, $Der^-(\boldsymbol{\tau}(\rho), \psi)$.
Then, by definition of $\mathsf{ApproxWrite}^-$ in Def. 20, $\psi \in \mathsf{ApproxWrite}^-(\boldsymbol{\tau}(\rho), \Psi)$, which yields a contradiction.
This establishes the theorem. □

Theorem 3. *Let $\boldsymbol{\tau}(\rho)$ be a transition class, and let $\tau \in \boldsymbol{\tau}(\rho)$. Then:*

$$\mathsf{Write}(\tau, \Psi) \quad \subseteq \quad \mathsf{ApproxWrite}(\boldsymbol{\tau}(\rho), \Psi)$$

Proof. Suppose the theorem is false, i.e. there exists a $\psi \in \Psi$ such that $\psi \in \mathsf{Write}(\tau, \Psi)$ and $\psi \notin \mathsf{ApproxWrite}(\boldsymbol{\tau}(\rho), \Psi)$.
Then, by definition of Write in Def. 9, $\psi \in \mathsf{Write}^+(\tau, \Psi)$ or $\psi \in \mathsf{Write}^-(\tau, \Psi)$.
Then, by Theorems 1 and 2, $\psi \in \mathsf{ApproxWrite}^+(\boldsymbol{\tau}(\rho), \Psi)$ or $\psi \in \mathsf{ApproxWrite}^-(\boldsymbol{\tau}(\rho), \Psi)$.
Then, by definition of ApproxWrite in Def. 20, $\psi \in \mathsf{ApproxWrite}^+(\boldsymbol{\tau}(\rho), \Psi)$, which yields a contradiction.
This establishes the theorem. □

5.4 Relations on Transitions

Having established a means to properly approximate write sets, in this section, we treat relations on transitions that are defined in terms of read and write sets, and on which POR and PBS algorithms rely. Before treating the relations, however, we first extend the notion of read sets to transition classes, as we need this in the remainder. We denote the read set of a transition class $\boldsymbol{\tau}(\rho)$ by $\mathsf{ApproxRead}(\boldsymbol{\tau}(\rho))$, and, because all transitions belonging to the same transition class have the same read set, define it as follows.

Definition 21. *Let $\boldsymbol{\tau}(\rho)$ be a transition class, and let $\tau \in \boldsymbol{\tau}(\rho)$.*
Then:

$$\mathsf{ApproxRead}(\boldsymbol{\tau}(\rho)) \quad = \quad \mathsf{Read}(\tau)$$

Although, there is nothing "approximate" about ApproxRead, we use the "Approx" prefix to synchronise the designation of read sets for transition classes with the designation of write sets for transition classes, i.e. $\mathsf{ApproxWrite}^+$, $\mathsf{ApproxWrite}^-$, and ApproxWrite.

We proceed with the relations. Recall in the following that $\boldsymbol{\Omega_T}$ is the set of all transition classes, that Ω_T is the set of all transitions, that $\mathcal{L}_\Psi$ is the language of mental state conditions, and that $\mathcal{L}_{\text{LTL}}$ is the set of LTL formulas.

5.4.1 Visibility

The visibility relation (e.g. [24]), denoted by $\mathsf{Visible} \subseteq \Omega_T \times \mathcal{L}_{\text{LTL}}$, expresses whether a transition can change the truth value of a mental state condition occurring in an LTL formula.

Definition 22. *Let ϕ be an LTL formula, let Ψ_ϕ be the set of* Mscs *occurring in ϕ, and let τ be a transition.*
Then:

$$\mathsf{Visible}(\tau, \phi) \quad \text{iff} \quad \mathsf{Write}(\tau, \Psi_\phi) \neq \emptyset$$

That is, $\mathsf{Visible}(\tau, \phi)$ is true if the write set of τ with respect to Ψ_ϕ is non-empty. In such case, we say that τ *is visible to* ϕ.

Next, we introduce the approximate visibility relation, denoted $\mathsf{ApproxVisible} \subseteq \boldsymbol{\Omega_T} \times \mathcal{L}_{\text{LTL}}$, that is defined for transition classes in terms of approximate write sets rather than in terms of precise write sets as the visibility relation above. This has the benefit that we can compute an over-approximative visibility relation before the transition system has been generated. Consequently, we can do much of the Por and Pbs algorithms' computations before the nested depth-first search commences, hence reducing computational overhead of Por and Pbs at runtime to a minimum.

Definition 23. *Let ϕ be an LTL formula, let Ψ_ϕ be the set of* Mscs *occurring in ϕ, and let $\boldsymbol{\tau}(\rho)$ be a transition class.*
Then:

$$\mathsf{ApproxVisible}(\boldsymbol{\tau}(\rho), \phi) \quad \text{iff} \quad \mathsf{ApproxWrite}(\boldsymbol{\tau}(\rho), \Psi_\phi) \neq \emptyset$$

That is, $\mathsf{ApproxVisible}(\boldsymbol{\tau}(\rho), \phi)$ is true if the approximate write set of $\boldsymbol{\tau}(\rho)$ with respect to Ψ_ϕ is non-empty. In such case, we say that $\boldsymbol{\tau}(\rho)$ *is visible to* ϕ.

The use of the approximate visibility relation does not impair correctness of Por and Pbs algorithms due to the following theorem, which states that if a transition τ is visible to a formula ϕ, then the transition class to which τ belongs is visible to ϕ as well.

Theorem 4. *Let $\boldsymbol{\tau}(\rho)$ be a transition class, and let $\tau \in \boldsymbol{\tau}(\rho)$.*
Then:

$$\mathsf{Visible}(\tau, \phi) \quad \Rightarrow \quad \mathsf{ApproxVisible}(\boldsymbol{\tau}(\rho), \phi)$$

1 *Proof.* By definition of Visible in Def. 22, $\mathsf{Write}(\tau, \Psi_\phi) \neq \emptyset$.
2 Then, by Theorem 3, $\mathsf{ApproxWrite}(\boldsymbol{\tau}(\rho), \Psi_\phi) \neq \emptyset$.
3 Then, by definition of ApproxVisible in Def. 23, $\mathsf{ApproxVisible}(\boldsymbol{\tau}(\rho), \phi)$. □

Why is correctness of Por and Pbs algorithms presented in subsequent chapters not impaired when approximate visibility is used? Well these algorithms are correct if we assume transitions visible, when they actually are not (though performance may suffer). In contrast, assuming transitions not visible, when they actually are, does violate correctness; the above theorem guarantees that this will never happen.

5.4.2 Enabled-by

The enabled-by relation (e.g. [24]), denoted by $\mathsf{EnabledBy} \subseteq \Omega_T \times \Omega_T$, expresses whether a transition $\tau \in \Omega_T$ can become enabled by execution of another transition $\tau' \in \Omega_T$. This happens if τ' changes the truth value of a mental state conditions on which enabled of τ depends is changed from false to true. That is, $\mathsf{EnabledBy}(\tau, \tau')$ is true if the positive write set of τ' with respect to the read set of τ is non-empty.[6] In such case, we say that τ *can be enabled by* τ'.

[6] We are only interested in the positive write set, because the negative write set contains only mental state conditions that become false. For example, if $\Psi = \{\psi, \neg\psi\}$ and $\mathsf{Write}^+(\tau', \Psi) = \{\neg\psi\}$ (i.e. the negation of ψ becomes true), then $\mathsf{Write}^-(\tau', \Psi) = \{\psi\}$.

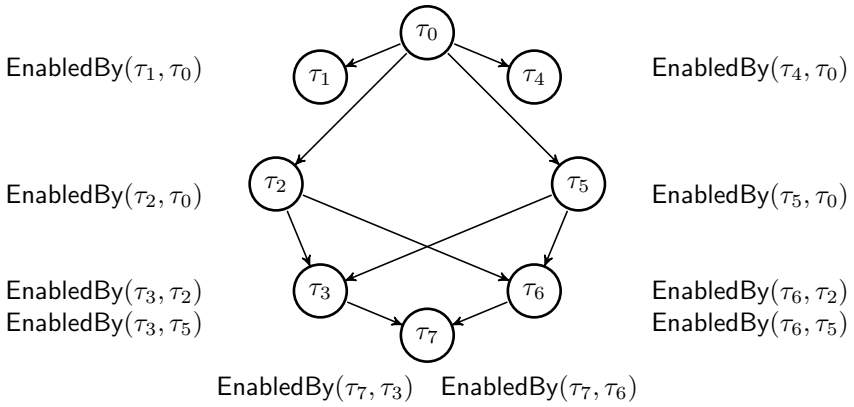


Figure 5.4: Enabled-by relation and digraph for `blenderAgent`.

Definition 24. *Let τ, τ' be transitions.*
Then:

$$\mathsf{EnabledBy}(\tau, \tau') \quad \text{iff} \quad \mathsf{Write}^{+}(\tau', \mathsf{Read}(\tau)) \neq \emptyset$$

We can regard the enabled-by relation as a digraph, in which vertices are transitions, and edges are elements of $\mathsf{EnabledBy}$. Specifically, if $\mathsf{EnabledBy}(\tau, \tau')$ is true, then the digraph contains an edge *from τ' to τ* (alternatively, one can regard the resulting graph as an *enables graph* that shows how transitions can enable each other).

To illustrate the previous, the enabled-by relation and digraph for `blenderAgent` are given in Fig. 5.4. For example, consider τ_6, which has as read set $\{\psi_3, \psi_8\}$ (see Table 5.3). None of the transitions have ψ_3 in their positive write set, whereas ψ_8 occurs in the positive write sets of τ_2 and τ_5. Hence, these transitions can make ψ_8 true, and as such enable τ_6. Consequently, $\mathsf{EnabledBy}(\tau_6, \tau_2)$ and $\mathsf{EnabledBy}(\tau_6, \tau_5)$.

Next, we introduce the approximate enabled-by relation, denoted $\mathsf{ApproxEnabledBy} \subseteq \boldsymbol{\Omega_T} \times \boldsymbol{\Omega_T}$, which is defined for transition classes in terms of approximate read and write sets rather than in terms of precise read and write set as the enabled-by relation above. This has the benefit that we can compute an over-approximative enabled-by relation before the transition system has been generated. Consequently, we can do much of the Por and Pbs algorithms' computations before the nested depth-first search commences, hence reducing computational overhead of Por and Pbs at runtime to a minimum.

Definition 25. *Let $\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')$ be transition classes.*
Then:

$$\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')) \quad \text{iff} \quad \mathsf{ApproxWrite}^{+}(\boldsymbol{\tau}(\rho'), \mathsf{ApproxRead}(\boldsymbol{\tau}(\rho)))$$

If $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$, we say that $\boldsymbol{\tau}(\rho)$ *can be enabled by* $\boldsymbol{\tau}(\rho')$.

The use of the approximate enabled-y relation does not impair correctness of Por and Pbs algorithms due to the following theorem, which states that if a transition τ can be enabled by execution of a transition τ', then the transition class to which τ belongs can be enabled by the transition class to which τ' belongs.

Theorem 5. *Let $\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')$ be transition classes, let $\tau \in \boldsymbol{\tau}(\rho)$, and let $\tau' \in \boldsymbol{\tau}(\rho')$.*
Then:

$$\mathsf{EnabledBy}(\tau, \tau') \quad \Rightarrow \quad \mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$$

Proof. By definition of $\mathsf{EnabledBy}$ in Def. 24, $\mathsf{Write}^+(\tau', \mathsf{Read}(\tau)) \neq \emptyset$.
Then, because $\mathsf{Read}(\tau) = \mathsf{ApproxRead}(\boldsymbol{\tau}(\rho))$ by Def. 21, $\mathsf{Write}^+(\tau', \mathsf{ApproxRead}(\boldsymbol{\tau}(\rho))) \neq \emptyset$.
Then, by Theorem 1, $\mathsf{ApproxWrite}^+(\boldsymbol{\tau}(\rho'), \mathsf{ApproxRead}(\boldsymbol{\tau}(\rho))) \neq \emptyset$.
Then, by definition of $\mathsf{ApproxEnabledBy}$ in Def. 25, $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$. □

Why is correctness of Por and Pbs algorithms presented in subsequent chapters not impaired when approximate enabled-by is used? Well these algorithms are correct if we assume transitions able to enable other transitions, when they actually are not (though performance of Por and Pbs may suffer). In contrast, assuming transitions not able to enable transitions, when they actually are, does violate correctness of these algorithms; the above theorem guarantees that this will never happen.

5.4.3 Independence

The independence relation (e.g. [24]) is a symmetric, anti-reflexive relation denoted by $\mathsf{Indep} \subseteq \Omega_T \times \Omega_T$. Informally, two transitions $\tau, \tau' \in \Omega_T$ are independent if execution of these transitions in either order results in the same successor state. More specifically, $\mathsf{Indep}(\tau, \tau')$ is true if for each mental state $\mu \in \Omega_M$ the following two conditions hold:

(i) *Enabledness*: if $\tau, \tau' \in \mathsf{En}(\mu)$ then $\tau \in \mathsf{En}(\tau'(\mu))$.

(ii) *Commutativity*: if $\tau, \tau' \in \mathsf{En}(\mu)$ then $\tau(\tau'(\mu)) = \tau'(\tau(\mu))$.

The enabledness condition states that a pair of independent transitions cannot *disable* each other. Note that the definition makes use of the fact that Indep is symmetric. The commutativity condition formalises that executing independent transitions in either order results in the same successor state. Two transitions are *dependent* if they are not independent. This is formalised by the *dependence relation* Dep, which is defined as the complement of Indep, i.e. $\mathsf{Dep} = (\Omega_T \times \Omega_T) \setminus \mathsf{Indep}$. Note that the dependence relation is symmetric and reflexive (because Indep is symmetric and anti-reflexive). The following definitions summarise the previous.

Definition 26. *Let τ, τ' be transitions, and let Ω_M be the set of all mental states. Then:*

$$\mathsf{Indep}(\tau, \tau') \quad \text{iff} \quad \text{for all } \mu \in \Omega_M : \begin{array}{l} \bullet \text{ if } \tau, \tau' \in \mathsf{En}(\mu) \text{ then } \tau \in \mathsf{En}(\tau'(\mu)) \\ \bullet \text{ if } \tau, \tau' \in \mathsf{En}(\mu) \text{ then } \tau(\tau'(\mu)) = \tau'(\tau(\mu)) \end{array}$$

Definition 27. *Let τ, τ' be transitions, and let Ω_M be the set of all mental states. Then:*

$$\mathsf{Dep}(\tau, \tau') \quad \text{iff} \quad \neg\mathsf{Indep}(\tau, \tau')$$

Similar to the enabled-by relation, we can regard the independence and dependence relations as graphs, in which vertices are transitions, and edges are elements of these relations. A difference is that these graphs are undirected (hence not digraphs), because both Indep and Dep are symmetric. The independence and dependence graphs for `blenderAgent` are given as an example in Fig. 5.5. All transitions of `blenderAgent` are mutually independent from each other, but they are dependent on themselves (by reflexivity of Dep).

Unfortunately, in practice, it is not possible to get an exact definition of Indep, because, to paraphrase [46], it is not practical to check the two conditions listed above in all mental states for all transitions. Therefore, as usual (e.g. [24]), we approximate the independence relation with a heuristic. We call a heuristic correct if all pairs in the independence relation that it yields satisfy the independence conditions. We do not require a heuristic to yield the *largest* independence relation possible: the estimation may be an under-approximation. Consequently, because the dependence relation is the complement of the independence relation, the dependence relation may

be an over-approximation. Although the POR algorithm in Chap. 7 is correct when Indep and Dep are under- and over-approximated (but not vice versa), their performance is better when the relations are more precise.

Informally, the reason that these approximations are allowed is that the POR algorithm will only reduce the state space in case two independent transitions are enabled simultaneously, in which case only one of them is executed. If these transitions are dependent, then both of them are executed, similar to model checking without POR. As such, assuming transitions dependent when they are actually are independent cannot cause certain computations on which correctness of the property under investigation depends to be pruned from the transition system. In contrast, assuming transitions independent when they actually are dependent may cause one of these transitions to not be executed. Consequently, a set of computations is pruned from the transition system, which may lead to different model checking results. Thus: we under-approximate the independence relation and over-approximate the dependence relation. (We simplified the previous exposition a bit; there are some additional requirements for transitions to actually be withheld from execution, which we treat in detail in Chap. 7.)

For (concurrent) imperative systems, various correct heuristics for approximating the independence relation are known (e.g. [24, 46, 96]). Central to these heuristics is the notion of system variables, which are, as discussed earlier, comparable to the type of variables one accesses and mutates in programming languages like Java and C. GOAL has no concepts that are similar to system variables in a straightforward fashion. One could regard the knowledge base K, the belief base Σ, and the goal base Γ as system variables, but the (in)dependence relations that one arrives at when regarding these entities as such using the existing heuristics is unsatisfactory. For example, consider the heuristic given in [24]:

> "Pairs of transitions that share a variable, which is changed by at least one of them, are dependent."

Now, if we would regard K, Σ, and Γ as system variables, then, for example, every transition that adds or deletes a fact from Σ would be dependent with all transitions that read the belief base (by means of mental state conditions), regardless of what the addition or deletion comprises, and regardless of which specific beliefs are read. This is too coarse in practice, i.e. we throw away too much information about the specific operations on Σ.

To take this information into account, we treat mental state conditions as if they are (boolean) system variables (see also Sect. 5.2.3). That is, we proceed with independence heuristics tailored to GOAL that are built on top of previously introduced theory of read and write sets, and as such make use of as much information about the specific operations on the mental state as possible. We start with a presentation of an independence heuristic that is defined in terms of single transitions (similar to the enabledness and commutativity conditions above). Subsequently, we extend this heuristic to transition classes such that we can compute the resulting approximate (in)dependence relations before the transition system is generated (similar to the introduction of the approximate visibility and enabled-by relations in previous subsections).

Heuristic for transitions

The independence heuristic for transitions, denoted $\mathrm{H}_{\mathsf{Indep}}$, is tailored to GOAL in the sense that mental state conditions and the add/del function Δ play a dominant role. $\mathrm{H}_{\mathsf{Indep}}$ consists of two sub-heuristics: one for the enabledness condition, denoted $\mathrm{H}^{en}_{\mathsf{Indep}}$, and another for the commutativity condition, denoted $\mathrm{H}^{com}_{\mathsf{Indep}}$. Let τ, τ' be transitions. The intuition behind $\mathrm{H}^{en}_{\mathsf{Indep}}$ is that if τ' cannot disable τ, then τ' cannot make the mental state conditions on which enabledness of τ depends false (recall that the enabledness condition states that two independent transition cannot disable each other). That is, the negative write set of τ' with respect to the read set of τ is empty. Formally, we can write $\mathrm{H}^{en}_{\mathsf{Indep}}$ as follows:

$\mathrm{H}^{en}_{\mathsf{Indep}}$**:** $\mathsf{Write}^{-}(\tau', \mathsf{Read}(\tau)) = \emptyset$

The intuition behind $\mathrm{H}^{com}_{\mathsf{Indep}}$ is that if τ and τ' do not add or delete the same belief or goal, then it does not matter in which order these changes are brought about. In contrast, for example, if τ adds a Pl fact χ to the belief base, while τ' deletes the same χ, then the order is of relevance: if first τ and then τ' is executed, χ is not in the belief base of the resulting mental state, whereas if first τ' and then τ is executed, χ does occur in the belief base. Thus, for two transitions to satisfy the commutativity transition, the additions and deletions of τ must be disjoint from the deletions and additions of τ'. Formally, we can write $\mathrm{H}^{com}_{\mathsf{Indep}}$ as follows. Let $\Delta(t) = \langle\langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma\rangle, \langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma\rangle\rangle$ for all $t \in \tau$, and let $\Delta(t') = \langle\langle \mathsf{Add}'_\Sigma, \mathsf{Add}'_\Gamma\rangle, \langle \mathsf{Del}'_\Sigma, \mathsf{Del}'_\Gamma\rangle\rangle$ for all $t' \in \tau'$.

$\mathrm{H}^{com}_{\mathsf{Indep}}$**:** $\mathsf{Add}_\Sigma \cap \mathsf{Del}'_\Sigma = \emptyset$ and $\mathsf{Del}_\Sigma \cap \mathsf{Add}'_\Sigma = \emptyset$ and $\mathsf{Add}_\Gamma \cap \mathsf{Del}'_\Gamma = \emptyset$ and $\mathsf{Del}_\Gamma \cap \mathsf{Add}'_\Gamma = \emptyset$.

The foundation for establishing correctness of $\mathrm{H}_{\mathsf{Indep}}$ is laid in Appx. C.2.1. That is, Lemmas 7,8 establish that if two transitions satisfy the sub-heuristics $\mathrm{H}^{en}_{\mathsf{Indep}}$ and $\mathrm{H}^{com}_{\mathsf{Indep}}$, respectively, then these transitions also satisfy the enabledness and commutativity conditions. Based on these results, Theorem 6 states that if two transitions satisfy $\mathrm{H}_{\mathsf{Indep}}$, then they are independent.

Theorem 6. *Let τ, τ' be transitions that satisfy $\mathrm{H}_{\mathsf{Indep}}$.*
Then:

$$\mathsf{Indep}(\tau, \tau')$$

Proof. Because τ and τ' satisfy $\mathrm{H}_{\mathsf{Indep}}$, they satisfy $\mathrm{H}^{en}_{\mathsf{Indep}}$ and $\mathrm{H}^{com}_{\mathsf{Indep}}$.
Then, by Lemmas 7,8, τ and τ' satisfy the enabledness and commutativity conditions.
Then, by definition of Indep in Def. 26, $\mathsf{Indep}(\tau, \tau')$.
This establishes the theorem. □

In addition to the previous, we prove a second theorem that states that if two transitions are dependent, then they violate $\mathrm{H}_{\mathsf{Indep}}$.

Theorem 7. *Let τ, τ' be transitions such that $\mathsf{Dep}(\tau, \tau')$.*
Then:

$$\tau, \tau' \text{ violate } \mathrm{H}_{\mathsf{Indep}}$$

Proof. By Theorem 6, $A \Rightarrow B$ in which: $A = [\tau, \tau'$ satisfy $\mathrm{H}_{\mathsf{Indep}}]$ and $B = \mathsf{Indep}(\tau, \tau')$
Also, by contraposition of $\Rightarrow$, $\neg B \Rightarrow \neg A$.
Then, by definition of Dep in Def. 27, if $\mathsf{Dep}(\tau, \tau')$ then τ, τ' violate $\mathrm{H}_{\mathsf{Indep}}$.
This establishes the theorem. □

Figure 5.6 shows the independence and dependence graphs when we compute the corresponding relations with $\mathrm{H}_{\mathsf{Indep}}$. Comparing these graphs with the graphs in Fig. 5.5, it is quite well observable that the independence and dependence relations are under-approximated and over-approximated, respectively, when using $\mathrm{H}_{\mathsf{Indep}}$. Specifically, we observe that τ_7 is now marked dependent with all other transitions. Additionally, τ_1 and τ_2, τ_4 and τ_5, and τ_3 and τ_6 are marked dependent. The reason is that although the transitions in these pairs can make a mental state condition on which enabledness of the other transition in the pair relies false, these transitions are never enabled simultaneously in practice. For example, τ_1 and τ_2 are only enabled after each other and never at the same time, i.e. in the same mental state (see Fig. 5.3). As such, they are independent according to the enabledness and commutativity conditions, but the approximative heuristic approach does not recognise this and marks them dependent.

Heuristic for transition classes

Next, we extend heuristic $\mathrm{H}_{\mathsf{Indep}}$ to transition classes. This has the benefit that we can compute an under-approximative independence relation and an over-approximative dependence relation before the transition system has been generated. Consequently, we can do much of the POR algorithm's computations before the nested depth-first search commences, hence reducing computational overhead of POR at runtime to a minimum.

We introduce heuristic $\mathrm{H}_{\mathsf{ApproxIndep}}$, which again consists of two sub-heuristics, namely $\mathrm{H}^{en}_{\mathsf{ApproxIndep}}$ and $\mathrm{H}^{com}_{\mathsf{ApproxIndep}}$. These sub-heuristics are similar to the sub-heuristics of $\mathrm{H}_{\mathsf{Indep}}$, although the definition of $\mathrm{H}^{com}_{\mathsf{ApproxIndep}}$ is slightly more involved because the sets computed by $\mathsf{Approx}\Delta$ may contain non-ground facts, while the sets computed by Δ contain ground facts only. Let $\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')$ be transition classes. $\mathrm{H}^{en}_{\mathsf{ApproxIndep}}$ states that the negative write set of $\boldsymbol{\tau}(\rho')$ with respect to the read set of $\boldsymbol{\tau}(\rho)$ is empty (similar to $\mathrm{H}^{en}_{\mathsf{Indep}}$).

$\mathrm{H}^{en}_{\mathsf{ApproxIndep}}$**:** $\mathsf{ApproxWrite}^{-}(\boldsymbol{\tau}(\rho'), \mathsf{ApproxRead}(\boldsymbol{\tau}(\rho))) = \emptyset$

The second sub-heuristic, i.e. $\mathrm{H}^{com}_{\mathsf{ApproxIndep}}$, states that the beliefs and goals that transitions belonging to $\boldsymbol{\tau}(\rho)$ may add to and remove from the belief and goal bases are disjoint from those that transitions belonging to $\boldsymbol{\tau}(\rho')$ may add and remove (similar to $\mathrm{H}^{com}_{\mathsf{Indep}}$). In the definition below, let $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_{\Sigma}, \mathsf{ApproxAdd}_{\Gamma}\rangle, \langle \mathsf{ApproxDel}_{\Sigma}, \mathsf{ApproxDel}_{\Gamma}\rangle\rangle$, and let $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho')) = \langle\langle \mathsf{ApproxAdd}'_{\Sigma}, \mathsf{ApproxAdd}'_{\Gamma}\rangle, \langle \mathsf{ApproxDel}'_{\Sigma}, \mathsf{ApproxDel}'_{\Gamma}\rangle\rangle$; recall from Sect. 2.3.1 that for notational convenience, we regard a conjunction of facts (i.e. a goal) as a set containing all conjuncts (and nothing else).

$\mathrm{H}^{com}_{\mathsf{ApproxIndep}}$**:**

- For all $\chi \in \mathsf{ApproxAdd}_{\Sigma}$ and $\chi' \in \mathsf{ApproxDel}'_{\Sigma}$, χ and χ' do not unify.
- For all $\chi \in \mathsf{ApproxDel}_{\Sigma}$ and $\chi' \in \mathsf{ApproxAdd}'_{\Sigma}$, χ and χ' do not unify.
- For all $\gamma \in \mathsf{ApproxAdd}_{\Gamma}$ and $\gamma' \in \mathsf{ApproxDel}'_{\Gamma}$ and $\chi \in \gamma$ and $\chi' \in \gamma'$, χ and χ' do not unify.
- For all $\gamma \in \mathsf{ApproxDel}_{\Gamma}$ and $\gamma' \in \mathsf{ApproxAdd}'_{\Gamma}$ and $\chi \in \gamma$ and $\chi' \in \gamma'$, χ and χ' do not unify.

The foundation for establishing correctness of $\mathrm{H}_{\mathsf{ApproxIndep}}$ is laid in Appx. C.2.2. That is, Lemmas 9,10 establish that if two transitions satisfy the sub-heuristics $\mathrm{H}^{en}_{\mathsf{ApproxIndep}}$ and $\mathrm{H}^{com}_{\mathsf{ApproxIndep}}$, respectively, then all transitions belonging to them also satisfy $\mathrm{H}^{en}_{\mathsf{Indep}}$ and $\mathrm{H}^{com}_{\mathsf{Indep}}$. Based on these results, Theorem 8 states that if two transition classes satisfy $\mathrm{H}_{\mathsf{ApproxIndep}}$, then all transitions belonging to them are independent. Additionally, Theorem 9 states that if two transitions are dependent, then the transition classes to which they belong violate $\mathrm{H}_{\mathsf{ApproxIndep}}$; this latter theorem is an implication of Theorem 8.

Theorem 8. *Let $\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')$ be transition classes that satisfy $\mathrm{H}_{\mathsf{ApproxIndep}}$.*
Additionally, let $\tau \in \boldsymbol{\tau}(\rho)$, and let $\tau' \in \boldsymbol{\tau}(\rho')$.
Then:

$$\mathsf{Indep}(\tau, \tau')$$

Proof. Because $\boldsymbol{\tau}(\rho)$, $\boldsymbol{\tau}(\rho')$ satisfy $\mathrm{H}_{\mathsf{ApproxIndep}}$, they satisfy $\mathrm{H}^{en}_{\mathsf{ApproxIndep}}$ and $\mathrm{H}^{com}_{\mathsf{ApproxIndep}}$.
Then, by Lemmas 9,10, τ and τ' satisfy $\mathrm{H}^{en}_{\mathsf{Indep}}$ and $\mathrm{H}^{com}_{\mathsf{Indep}}$.
Then, by Theorem 6, $\mathsf{Indep}(\tau, \tau')$.
This establishes the theorem. □

In addition to the previous, we prove a second theorem that states that if two transitions are dependent, then the transition classes to which they belong violate $\mathrm{H}_{\mathsf{ApproxIndep}}$.

Theorem 9. *Let* $\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')$ *be transition classes such that there exist a* $\tau \in \boldsymbol{\tau}(\rho)$ *and* $\tau' \in \boldsymbol{\tau}(\rho')$ *such that* $\mathsf{Dep}(\tau, \tau')$.
Then:

$$\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho') \text{ violate } \mathrm{H}_{\mathsf{ApproxIndep}}$$

Proof. By Theorem 8, $A \Rightarrow B$ in which: $A = [\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')$ satisfy $\mathrm{H}_{\mathsf{ApproxIndep}}]$ and $B = [\mathsf{Indep}(\tau, \tau')$ for all $\tau \in \boldsymbol{\tau}(\rho)$ and $\tau' \in \boldsymbol{\tau}(\rho')]$.
Also, by contraposition of $\Rightarrow$, $\neg B \Rightarrow \neg A$.
Then, by definition of Dep in Def. 27, if there exist a $\tau \in \boldsymbol{\tau}(\rho)$ and $\tau' \in \boldsymbol{\tau}(\rho')$ such that $\mathsf{Dep}(\tau, \tau')$ then $\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')$ violate $\mathrm{H}_{\mathsf{ApproxIndep}}$.
This establishes the theorem. □

Figure 5.7 shows the independence and dependence graphs when we compute the corresponding relations with $\mathrm{H}_{\mathsf{ApproxIndep}}$. We observe that the independence relation is under-approximated quite severely compared to Figs. 5.5,5.7 (hence the dependence relation over-approximated). In fact, only eight transitions are marked independent, while actually all transitions are independent with each other. Nevertheless, we *must* use $\mathrm{H}_{\mathsf{ApproxIndep}}$ in the remainder, because it can be computed before the transition system is generated, in contrast to $\mathrm{H}_{\mathsf{Indep}}$.

We summarise the previous by explicitly introducing the *approximate (in)dependence relations* on transition classes, in the same spirit as the approximate visibility relation (see Def. 23) and the approximate enabled-by relation (see Def. 25).

Definition 28. *Let* $\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')$ *be transition classes.*
Then:

$$\mathsf{ApproxIndep}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')) \quad \text{iff} \quad \mathrm{H}_{\mathsf{ApproxIndep}} \text{ is satisfied}$$

Definition 29. *Let* $\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')$ *be transition classes.*
Then:

$$\mathsf{ApproxDep}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')) \quad \text{iff} \quad \neg\mathsf{ApproxIndep}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$$

Subsequently, the next theorems follow immediately from Theorems 8,9.

Theorem 10. *Let* $\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')$ *be transition classes, let* $\tau \in \boldsymbol{\tau}(\rho)$, *and let* $\tau \in \boldsymbol{\tau}(\rho')$.
Then:

$$\mathsf{ApproxIndep}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')) \quad \Rightarrow \quad \mathsf{Indep}(\tau, \tau')$$

Theorem 11. *Let* $\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')$ *be transition classes, let* $\tau \in \boldsymbol{\tau}(\rho)$, *and let* $\tau \in \boldsymbol{\tau}(\rho')$.
Then:

$$\mathsf{Dep}(\tau, \tau') \quad \Rightarrow \quad \mathsf{ApproxDep}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$$

From Theorem 11, it follows that the approximate dependence relation ApproxDep is reflexive like the dependence relation Dep. To see this, observe that we have $\mathsf{Dep}(\tau, \tau)$ for any transition τ. Let $\boldsymbol{\tau}(\rho)$ be the transition class to which τ belongs. Then, by Theorem 11, $\mathsf{ApproxDep}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho))$. Consequently, if the approximate dependence relation is used, all transitions belonging to a transition class are deemed interdependent. We will use this characteristic of the approximate dependence relation in Chap. 7.

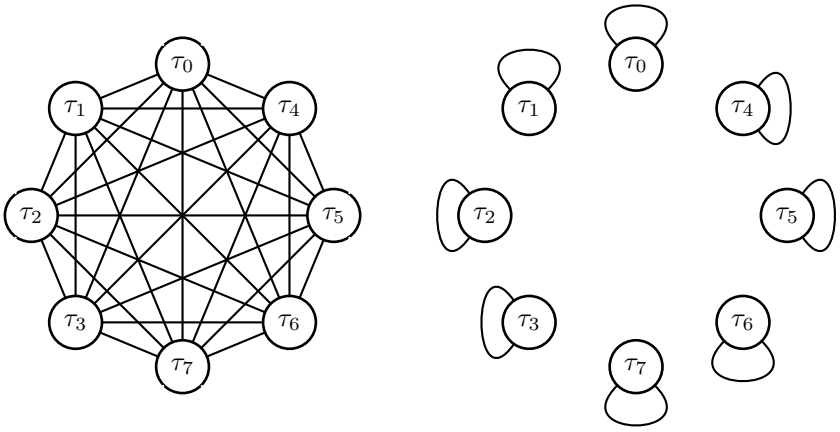


Figure 5.5: Independence graph (left) and dependence graph (right) for `blenderAgent`.

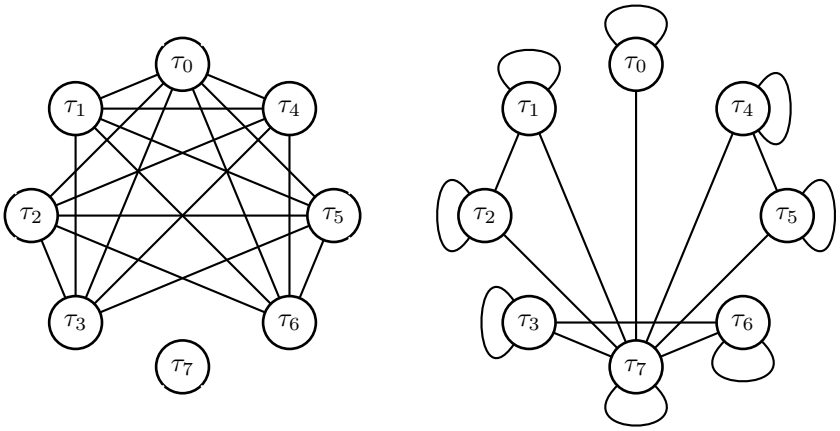


Figure 5.6: Independence graph (left) and dependence graph (right) for `blenderAgent` when using $\mathrm{H}_{\mathsf{Indep}}$.

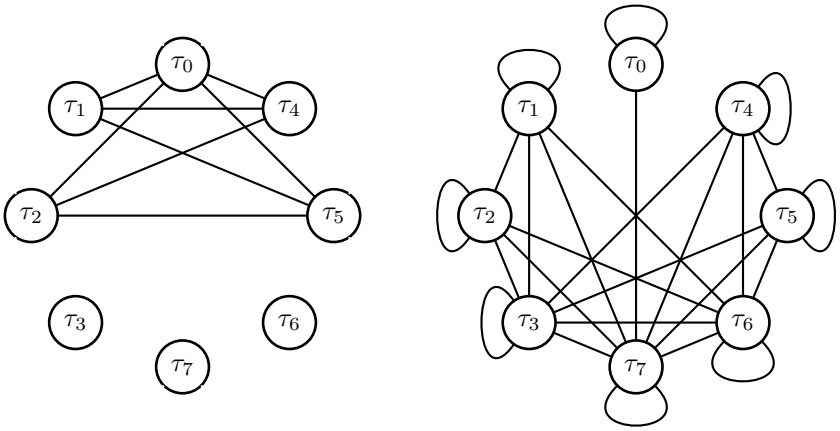


Figure 5.7: Independence graph (left) and dependence graph (right) for `blenderAgent` when using $\mathrm{H}_{\mathsf{ApproxIndep}}$.

5.5 Summary

In this chapter we introduced a transition theory for Goal according to the Por notion of transitions: one in which transitions can be thought of as operations on mental states rather than as mental state pairs in the transition system. The first section of the chapter elaborated on this concept and its formal definition. The two sections that followed concerned read and write sets: in Sect. 5.2, their definitions were presented, and in Sect. 5.3, we elaborated on their computation in practice. Finally, in the last section, we defined several relations on transitions that are used by the Por and Pbs algorithms presented in the next chapters. Central to most of the chapter was the observation that we can only approximate write sets of transitions (and the relations defined in terms of them) because at the time that we want to compute them, typically before the transition system is generated, we have insufficient information for a precise computation. Except for the independence relation, all approximations in this chapter were over-approximation (rather than under-approximations). In proofs of correctness of the Pbs and Por algorithms, we will use these over-approximative characteristics.

Chapter 6

Property-Based Slicing

Suppose we want to verify whether `blenderAgent` never believes that it has put more bananas into the blender's reservoir than prescribed by the recipe. We can specify this property formally in LTL as $\varphi = \mathsf{G}\,\texttt{bel(recipe(bananas,Qr),added(bananas,Q),Q=<Qr)}$. As remarked in Sect. 2.5, `blenderAgent` has 16 computations, each one corresponding to a distinct path through its transition system $\mathcal{T}$; the model checker needs to investigate all of them to verify φ. Now, let us have a closer look at Fig. 5.3, in which $\mathcal{T}$ appears with symbolic names for transitions, combined with Tables 5.1 and 5.2. One might notice that at least transitions τ_4 and τ_5 do not really matter for the verification of φ: whether oranges are added does not influence how many bananas are added, and can as such not influence the truth or falsehood of φ. Nevertheless, the presence of these transitions increases the size of $\mathcal{T}$ and consequently the number of computations to investigate.

Ideally, we would like to verify the reduced transition system given in Fig. 6.1 in which only transitions τ_0, τ_1, and τ_2 have been preserved. That is, we want to preserve sufficiently many behaviours of `blenderAgent` to establish that the property is false (if it is false); all other transitions may be *sliced away*. As a result, in this example only one computation remains, enabling the verification procedure to finish much faster. In this chapter, we further investigate this idea of disregarding transitions that are not relevant for the property at hand. This technique is called *property-based slicing* (PBS) or simply *slicing*.

The chapter is organised as follows. In Sect. 6.1, we give a brief introduction to slicing, particularly with respect to its use in model checking. Section 6.2 treats the concept of *influence*, which is what our slicing algorithm is based on. The actual algorithm is presented in Sect. 6.3, and in Sect. 6.4, we assess its effectiveness based on three case studies. Section 6.5 discusses possible extensions to the basic algorithm, and Sect. 6.6 concludes the chapter.

6.1 Slicing and Model Checking

The idea of slicing programs was introduced in [104]. In general, slicing concerns the removal of parts of a program that are not relevant for the analysis at hand. Applications of slicing range from debugging to reverse engineering; a detailed overview of slicing techniques for imperative programming languages is given in [95]. In model checking, as outlined above, slicing is used to remove the parts of the program that do *not* influence the truth or falsehood of a property φ in the hope that the transition system of the program under investigation is reduced. In contrast, the parts that *can* influence the truth value of φ must be preserved at all costs, because otherwise, the model checking results obtained for the sliced program cannot be used to establish truth or falsehood of φ for the original program.

With respect to model checking, slicing algorithms have for example been implemented in the earlier mentioned model checkers SPIN (e.g. [75]) and JPF (e.g. [103]). Slicing efforts in the agent verification community have up to now been limited to a single publication: in [13], a slicing algorithm for AGENTSPEAK agents is presented based on the work of [108] in which logic programs are sliced. The authors of [13] chose [108] as a basis for their own algorithm, because *plans* in

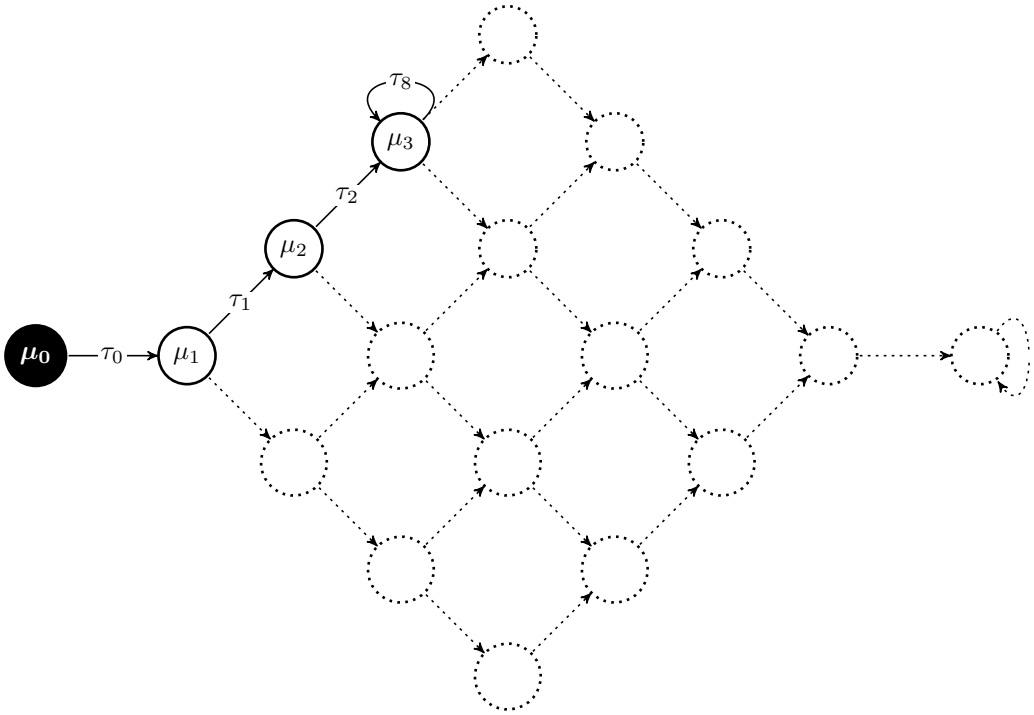


Figure 6.1: Ideal transition system of `blenderAgent` when verifying the property φ = `Gbel(recipe(bananas,Qr),added(bananas,Q),Q=<Qr)`.

AgentSpeak are similar to guarded clauses; [108] provides a slicing algorithm for logic programs that are represented as a collection of such guarded clauses. Consequently, the algorithm presented in [13] slices AgentSpeak agents by removing plans from the plan base of such agents. However, because Goal is not equipped with plan-like constructs like AgentSpeak, the method of [13] cannot easily be adapted to slice Goal agents. We have therefore decided to build our slicing algorithm from the ground, tailored to Goal.

Of course, although designed specifically for Goal, our algorithm bears similarities with existing slicing algorithms in the literature as well. First, slicing is an operation that is carried out *before* the actual model checking procedure commences to minimise overhead during verification. As such, slicing methods (including ours) work by static analysis of the code by which a program (in our case a Goal agent) is defined. During this analysis, the slicing algorithm transforms the original program (Goal agent) to the sliced program with respect to a *slicing criterion* (e.g. [95]). Informally, a slicing criterion is a representation of the points of interest of a program, e.g. the value of a particular system variable of an imperative program at some point during execution. When slicing is applied to model checking, the slicing criterion is extracted from the property under investigation (e.g. [13]). Hence, *property-based* slicing.

Central to many slicing algorithms is that they represent the program under investigation as a graph. Subsequently, the slicing problem can be formulated as a graph reachability problem, provided that the slicing criterion can be encoded by vertices in the graph. The use of graph representations of programs in slicing algorithms was pioneered in [77]. With respect to model checking, it has, for example, been used in [22, 102] for slicing hardware specifications, in [75, 103] for slicing Promela and Java (combined with, respectively, Spin and Jpf), and in [13] for slicing AgentSpeak. We use such a graph approach as well.

In broad lines, thus, the slicing algorithm that we present is similar to existing algorithms. However, besides the fact that slicing algorithms are language dependent such that a Goal slicing algorithm is in that respect novel, our approach differs also with respect to the angle from which we

approach the slicing problem. That is, whereas slicing algorithms in the literature focus exclusively on the code by which a program is defined, we take our transition theory presented in the previous chapter as a formal basis. Such an approach to slicing has, as far as we know, not been taken in the literature, whereas in our experience, it enables proofs of correctness to be both clearer and cleaner (a preliminary version of the algorithm we present was based exclusively on source code, which required elaborate proofs). Other benefits of basing PBS on the transition theory concern the synergy that is obtained when PBS is combined with partial order reduction. We come back to this in more detail in Sect. 7.4.

Let P be the GOAL agent under investigation. During the execution of the PBS algorithm, P is transformed to its sliced version $\overline{P}$. We denote this transformation by the function Slice, which, in addition to P, takes an LTL formula ϕ as input (i.e. the slicing criterion). Typically, this formula is the negated property under investigation. We will later prove that there exists a subset of $\mathcal{L}_{\text{LTL}}$ whose each formula ϕ satisfies the following: if P has a computation that satisfies ϕ, then at least one such computation is preserved in $\overline{P} = \mathsf{Slice}(P, \phi)$. Thus, if the negated property $\neg\varphi$ is in this fragment of $\mathcal{L}_{\text{LTL}}$, then we may model check $\overline{P} = \mathsf{Slice}(P, \neg\varphi)$ rather than P.

Informally, the idea is to only preserve those action rules of P that generate at least one transition that can *influence* the truth or falsehood of the formula ϕ. That is, rather than slicing individual transitions, we slice entire transition classes (transition classes were introduced in Sect. 5.3). There are two ways for a transition τ to influence ϕ: (i) τ is *visible* to ϕ, i.e. it can change the truth value of at least one MSC occurring in ϕ, or (ii) τ can *enable* another transition that influences ϕ. Thus, our slicing algorithm relies on the computation of the approximate visibility and enabled-by relations, which generalise over all transitions belonging to a transition class (see Sects. 5.4.1 and 5.4.2).

6.2 Influence

In this section, we more formally discuss the notion of *influence*. We start with a graph-theoretic account, which forms the basis of the algorithm in Sect. 6.3, and proceed with a more formal exposition, which is later used to prove the algorithm correct.

6.2.1 Influence Graph

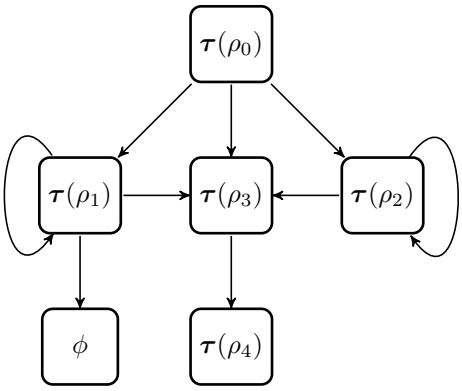
Let $\boldsymbol{\Omega_T}$ be the set of all transition classes of the agent P under investigation. Then, the *influence graph*, denoted $\mathcal{G}(\boldsymbol{\Omega_T}, \phi)$, is a digraph that captures how transitions in transition classes can influence an LTL formula ϕ. More specifically, $\mathcal{G}(\boldsymbol{\Omega_T}, \phi) = \langle \mathcal{V}, \mathcal{E} \rangle$ in which $\mathcal{V}$ is a set of vertices and $\mathcal{E}$ is a set of edges.

Definition 30. *Let $\boldsymbol{\Omega_T}$ be the set of transition classes, let $\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')$ be typical elements of $\boldsymbol{\Omega_T}$, let ϕ be an LTL formula, and let $\mathcal{G}(\boldsymbol{\Omega_T}, \phi) = \langle \mathcal{V}, \mathcal{E} \rangle$ be the influence graph with respect to ϕ. Then:*

$$
\begin{aligned}
\mathcal{V} &= \boldsymbol{\Omega_T} \cup \{\phi\} \\
\mathcal{E} &= \{\langle \boldsymbol{\tau}(\rho), \phi \rangle \in \boldsymbol{\Omega_T} \times \{\phi\} \mid \mathsf{ApproxVisible}(\boldsymbol{\tau}(\rho), \phi)\} \\
&\quad \cup \{\langle \boldsymbol{\tau}(\rho'), \boldsymbol{\tau}(\rho) \rangle \in \boldsymbol{\Omega_T} \times \boldsymbol{\Omega_T} \mid \mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))\}
\end{aligned}
$$

Informally, the vertices of the influence graph are the transition classes of P, plus a distinguished vertex for the formula ϕ, called the *formula vertex*. Then, for each transition class $\boldsymbol{\tau}(\rho)$ that contains a transition that is visible to ϕ, we draw an edge from $\boldsymbol{\tau}(\rho)$ to ϕ (upper part of the definition of $\mathcal{E}$). The graph is completed with a set of edges (lower part of the definition of $\mathcal{E}$) from each transition class that contains a transition that can enable the transition in another transition class. An example of `blenderAgent`'s influence graph with respect to $\phi = \mathsf{F}\,$`bel(recipe(bananas,Q), added(bananas,Q))` is given in Fig. 6.2.

A *route* through $\mathcal{G}(\boldsymbol{\Omega_T}, \phi)$ is a finite sequence of vertices $\boldsymbol{\tau}(\rho_0) \cdots \boldsymbol{\tau}(\rho_n)\phi$, abbreviated $\boldsymbol{\tau}(\rho_0) \rightsquigarrow \phi$, such that every transition class occurs only once on a route, i.e. for all $0 \leq i, j \leq n$: if $i \neq j$



ρ_0 = **if goal**(`filled`) $\wedge$ **bel**(`not(washed)`) **then** `wash`
ρ_1 = **if goal**(`filled`) $\wedge$ **bel**(`toAdd(bananas)`) **then** `add(bananas,1)`
ρ_2 = **if goal**(`filled`) $\wedge$ **bel**(`toAdd(oranges)`) **then** `add(oranges,1)`
ρ_3 = **if goal**(`filled`) $\wedge$ **bel**(`recipe(F,_)`) **then** `tick(F)`
ρ_4 = **if goal**(`switch(on)`) $\wedge$ **bel**(`filled`) **then** `blend`

Figure 6.2: Influence graph for `blenderAgent` with respect to the formula ϕ = `F bel(recipe(bananas,Q), added(bananas,Q))`

then $\rho_i \neq \rho_j$. Note that every route must end in ϕ. The length of a route $\boldsymbol{\tau}(\rho_0) \rightsquigarrow \psi$ is denoted by $|\boldsymbol{\tau}(\rho_0) \rightsquigarrow \psi|$, and we denote the set of all routes through the graph $\mathcal{G}(\boldsymbol{\Omega_T}, \phi)$ by $\mathit{Routes}(\mathcal{G}(\boldsymbol{\Omega_T}, \phi))$. Because routes must end in ϕ, the influence graph in Fig. 6.2 has two routes, namely $\boldsymbol{\tau}(\rho_0)\boldsymbol{\tau}(\rho_1)\psi$ and $\boldsymbol{\tau}(\rho_1)\psi$. We deliberately use the term "route" rather than "path", as we have associated the term "path" with sequences of mental states in the transition system of a Goal agent. Indeed, the definition of a path in that context is also quite different from the definition of a route that we associate with influence graphs.

Informally, every route through the influence graph represents one way in which transitions belonging to the transition class from which the route starts can influence ϕ. For example, the transitions belonging to $\boldsymbol{\tau}(\rho_1)$ of `blenderAgent` can directly influence ϕ as they might be visible to it (see Fig. 6.2). Additionally, the transitions belonging to $\boldsymbol{\tau}(\rho_0)$ can influence ϕ, because their execution may cause transitions in $\boldsymbol{\tau}(\rho_1)$ to become enabled. In contrast, all transition classes that are *not* on any route may be removed from the transition system as they cannot influence ϕ (this is not the case for all LTL formulas; we discuss this in more detail in the next subsection).

As mentioned in the previous section, representing programs (i.e. Goal agents) as a graph in which the slicing criterion (i.e. LTL formulas) can be represented such that slicing becomes a graph reachability problem is not novel. Such graphs are for example called *system dependence graph* (e.g. [22]), *program dependence graph* (e.g. [102]) or *literal dependence net* (e.g. [13]). We favour a name in which "dependence" does not occur, as the influence graph is not based on the dependence relation.

6.2.2 Influence Relation

Having given a graph-theoretic account of influence, we now proceed with a more formal exposition of the same concept. The reason for using two formalisms to express the same concept is that the graph-theoretic account is convenient from a conceptual and algorithmic point of view, while the relational account that we treat below turns out favourable when we want to prove correctness of the algorithm. For the latter to be meaningful, a correspondence need be proven between the two formalisms; we will do this later. First, we give the definition of the *influence relation*, denoted $\mathsf{Influence} \subseteq \boldsymbol{\Omega_T} \times \mathcal{L}_{\mathrm{LTL}} \times \mathbb{N}^+$.

Definition 31. *Let $\boldsymbol{\tau}(\rho)$ be a transition class, and let ϕ be an LTL formula. Then:*

$$\begin{array}{lll} \mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, 1) & \text{iff} & \mathsf{ApproxVisible}(\boldsymbol{\tau}(\rho), \phi) \\ \mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i+1) & \text{iff} & \mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho'), \boldsymbol{\tau}(\rho)) \text{ and } \mathsf{Influence}(\boldsymbol{\tau}(\rho'), \phi, i) \end{array}$$

Informally, the influence relation expresses whether a transition class $\boldsymbol{\tau}(\rho)$ influences ϕ from a *distance* i. If the distance is 1, we say that $\boldsymbol{\tau}(\rho)$ influences ϕ *directly*; otherwise, it influences ϕ *indirectly*. Direct influence corresponds to the approximate visibility relation, while indirect influence corresponds to the approximate enabled-by relation. Note that the same transition class may influence ϕ from several distances, i.e. it may be the case that $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i)$ and $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i')$ and $i \neq i'$. This will not cause problems: if a transition class influences the formula from multiple distances, this merely means that there exist multiple different routes in the influence graph from the transition class to the formula vertex. The correspondence that we will prove, however, only states that if there exists a route from a transition class $\boldsymbol{\tau}(\rho)$ to the formula vertex, then there exists an i such that $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i)$ is true, and vice versa. That is, the fact that there may me more routes and more i-s is not captured by this correspondence (and unnecessary for our purpose).

A concept closely related to the influence relation is the *influential set*, denoted $\mathsf{Influential}(\phi) \subseteq \Omega_T$.

Definition 32. *Let ϕ be an LTL formula and let $\boldsymbol{\Omega_T}$ be the set of all transition classes. Then:*

$$\mathsf{Influential}(\phi) \quad = \quad \bigcup_{\boldsymbol{\tau}(\rho) \in \{\, \boldsymbol{\tau}(\rho) \in \boldsymbol{\Omega_T} \mid \text{there exists an } i \geq 0 \text{ such that } \mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i) \,\}} \boldsymbol{\tau}(\rho)$$

That is, the influential set is a set containing all the transitions in transition classes that influence ϕ.

Next, we prove the correspondence between the influence graph and the influential set. The correspondence, given as Theorem 12, states that there exists a route from a transition class to the formula vertex iff the transitions belonging to this transition class are in the influential set. The proof is based on two lemmas that we prove in Appx. C.3.1. That is, Lemma 11 states that if $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi$ is a route through the influence graph, then $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i)$ is true for some $i \geq 1$, and Lemma 12 states that if $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i)$ is true, then there exists a route $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi$ through the influence graph. Both lemmas are proven by induction: the former on the length of a route $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi$, and the latter on the value of i in $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i)$. Given the similarities between the definition of the influence graph in Def. 30 (especially the set of edges $\mathcal{E}$) and the definition of the influence relation in Def. 31, these proofs are quite straightforward. Basically, theorem 12 summarises these two lemmas for influential sets, which are derived from the influence relation.

Theorem 12. *Let ϕ be an LTL formula, let $\boldsymbol{\Omega_T}$ be the set of transition classes, and let $\mathcal{G}(\boldsymbol{\Omega_T}, \phi) = \langle \mathcal{V}, \mathcal{E} \rangle$ be the corresponding influence graph. Then:*

$$\boldsymbol{\tau}(\rho) \rightsquigarrow \phi \in \mathit{Routes}(\mathcal{G}(\boldsymbol{\Omega_T}, \phi)) \quad \Leftrightarrow \quad \boldsymbol{\tau}(\rho) \subseteq \mathsf{Influential}(\phi)$$

Proof.

($\Rightarrow$) By Lemma 11, there exists an $i \geq 1$ such that $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i)$.
Then, by definition of $\mathsf{Influential}$ in Def. 32, $\boldsymbol{\tau}(\rho) \subseteq \mathsf{Influential}(\phi)$.

($\Leftarrow$) By definition of $\mathsf{Influential}$ in Def. 32, there exists a $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i)$.
Then, by Lemma 12, there exists a $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi \in \mathit{Routes}(\mathcal{G}(\boldsymbol{\Omega_T}, \phi))$.

This establishes the theorem. □

As aforementioned, the relevance of this theorem is that it enables us to base the algorithm's proof of correctness on the influential set, even though the algorithm operates on the influence graph.

6.2.3 LTL and Influential Sets

The transitions in the influential set $\mathsf{Influential}(\phi)$ are all the transitions that we want to preserve in the sliced agent and nothing more. However, as mentioned earlier, not all properties that we can specify in LTL are invariant to the removal of the transition classes whose transitions are not in $\mathsf{Influential}(\phi)$. The reason is that we make the tacit assumption that ϕ's truth depends only on transitions that can change the truth value of the MSCs that occur in ϕ. There are, however, two LTL operators for which this assumption does not hold: X and R. That is, X and R formulas can become true *without* the execution of a transition that changes the truth value of an MSC in the respective formula. We informally discuss this issue below (see also [57]).

X Suppose $\phi = \mathsf{X}\psi$ must be true in some state μ. Then, by the semantics of X, ψ must be true in μ's successor, say μ'. Suppose μ' can only be reached by executing a transition that is outside $\mathsf{Influential}(\phi)$. Still, ψ may be true in μ' such that $\mathsf{X}\psi$ is true in μ: this happens if ψ was already true in μ. Consequently, if we remove all transitions outside $\mathsf{Influential}(\phi)$, this computation on which $\mathsf{X}\psi$ is true does not exist in the sliced agent in contrast to the original agent. Subsequently, if $\mathsf{X}\psi$ is the negated property under investigation, then the model checker will incorrectly report that the original agent satisfies the property, because no counterexample is found in the transition system of the sliced agent.

R The problem with the R operator is that we defined it as *weak release* (see Sect. 3.1.1). This means that ϕ_1 does not need to be true eventually for $\phi = \phi_1 \,\mathsf{R}\, \phi_2$ to be true. Consequently, a (series of) transition(s) that are outside $\mathsf{Influential}(\phi)$ (and hence do not affect the truth of ϕ_2) may be executed until a previously visited state is encountered, i.e. a cycle is reached, such that $\phi_1 \,\mathsf{R}\, \phi_2$ is true. Now, if we remove all these transitions, such a cycle may not exist any more in the transition system of the sliced agent. Then, similar to the X case, the model checker will incorrectly report that there exists no counterexample.

Therefore, in the remainder of this chapter, we only consider LTL properties whose NNF is in the $\{\mathsf{X},\mathsf{R}\}$-free fragment of LTL. We abbreviate this fragment by LTL$\backslash\{\mathsf{X},\mathsf{R}\}$, and denote the set of all formulas it contains by $\mathcal{L}_{\text{LTL}}\backslash\{\mathsf{X},\mathsf{R}\}$. Though the restriction to LTL$\backslash\{\mathsf{X},\mathsf{R}\}$ formulas is unfortunate, a significant class of LTL properties is still covered. For example, all safety properties (which are, according to [102], the majority of the properties required to verify systems) can be verified in combination with our PBS algorithm. An additional assumption that we make is that all formulas under consideration contain at least one temporal operator. Otherwise, slicing is unnecessary, because the truth of a fully propositional formula can be established in the initial mental state, i.e. no transitions need be executed.

In Appx. C.3.2, we prove that if a GOAL agent P has a computation on which an $\mathcal{L}_{\text{LTL}}\backslash\{\mathsf{X},\mathsf{R}\}$ formula ϕ is satisfied, then P also has a computation on which ϕ is satisfied on a finite prefix that is generated solely by transition from $\mathsf{Influential}(\phi)$. We call this latter computation a *minimal ϕ-satisfying computation*. The existence of such a computation suggests that if we slice away all transitions that are not in $\mathsf{Influential}(\phi)$, sufficient transitions remain to establish ϕ's truth. Informally, the lemma that establishes the existence of such a computation, i.e. Lemma 13, is proven as follows. Suppose $\boldsymbol{\pi}$ is a computation on which an LTL$\backslash\{\mathsf{X},\mathsf{R}\}$ formula ϕ is satisfied, and suppose that we can establish this by inspecting the prefix of size k (e.g. $\phi = \mathsf{F}\psi$ and ψ is true for the first time in the k-th mental state on $\boldsymbol{\pi}$). Let T be the set of transitions that are executed during the generation of this k-prefix (i.e. prefix of size k) and that are *not* in $\mathsf{Influential}(\phi)$. Specifically, let τ be the last such transition executed. Now, we claim that there also exists a computation, say $\boldsymbol{\pi}'$, whose $(k-1)$-prefix is generated by the same sequence of transitions as the k-prefix of $\boldsymbol{\pi}$, except that τ is left out, and on which satisfaction of ϕ can be established.

The intuition is that because $\tau \notin \mathsf{Influential}(\phi)$, it cannot enable transitions in $\mathsf{Influential}(\phi)$ (otherwise, τ would be in $\mathsf{Influential}(\phi)$ itself). This is stated by Prop. 5 in Appx. C.3.2. As such, all transitions that are executed after τ on the k-prefix of $\boldsymbol{\pi}$ can also be executed (in the same order) if τ is not executed before them. Also, because $\tau \notin \mathsf{Influential}(\phi)$, execution of τ did not influence the truth value of any of the mental state conditions in ϕ (i.e. τ is not visible to

ϕ). Consequently, the formula that need be true before execution of τ is the same as the formula that need be true after its execution. In Appx. C.3.2, this is stated by Prop. 6. Basically, this proposition makes explicit that all LTL$\setminus\{\mathsf{X},\mathsf{R}\}$ formulas require the execution of transitions in the influential set to become true. As a consequence, the execution of τ did not influence the truth of ϕ, and thus can be omitted. To formalise and prove Prop. 6, we use the notion of *progression* (e.g. [2]): the progression of a formula ϕ in the i-th mental state $\boldsymbol{\pi}_i$ on a computation $\boldsymbol{\pi}$, denoted $\mathit{progression}(\phi, i, \boldsymbol{\pi})$ is the formula that must be true in $\boldsymbol{\pi}_i$ for ϕ to be satisfiable by $\boldsymbol{\pi}$. The formal definition of progression appears as Def. 33 in Appx. C.3.2. To summarise the previous, because $\tau \notin \mathsf{Influential}(\phi)$, not executing τ does not lead to the transitions succeeding τ on the k-prefix of $\boldsymbol{\pi}$ to become disabled, while ϕ's truth can still be established.

Thus, there exists a computation $\boldsymbol{\pi}'$ whose $(k-1)$-prefix is generated by the same sequence of transitions as the k-prefix of $\boldsymbol{\pi}$, except that τ is left out, and on which satisfaction of ϕ can be established. Next, we can eliminate the last transition outside $\mathsf{Influential}(\phi)$, say τ', executed to generate the $(k-1)$-prefix of $\boldsymbol{\pi}'$, yielding a computation $\boldsymbol{\pi}''$, whose $(k-2)$-prefix is generated by the same sequence of transitions as the $(k-1)$-prefix of $\boldsymbol{\pi}'$, except that τ' is left out, and on which satisfaction of ϕ can be established. That is, by induction, we can eliminate all j transitions outside $\mathsf{Influential}(\phi)$ that were executed to generate the k-prefix of $\boldsymbol{\pi}$ (with which we started) to gain a computation $\boldsymbol{\pi}^{(j)}$ with a $(k-j)$-prefix that is generated without execution of a transition outside $\mathsf{Influential}(\phi)$, and on which satisfaction of ϕ can be established. This is the minimal ϕ-satisfying computation that we are looking for; we refer to Appx. C.3.2 for details. In the eventual proof of correctness of our algorithm, this results is used as a building block.

6.3 Algorithm

We have developed sufficient theory to formulate our slicing algorithm. As discussed, the influence graph and influence relation / influential set are used for two different purposes: the influence graph is the structure on which our algorithm operates, while the influence relation / influential set is used to prove the algorithm correct. The latter is possible due to correspondence between the influence graphs and influential sets as stated by Theorem 12 in Appx. C.3.1. We first present the algorithm, i.e. Alg. 3, and later concern ourselves with its correctness.

The algorithm works as follows. On line 1, a new rule base for the sliced agent is initialised as empty. On lines 2-6, the rules in the original rule base are iterated, and only if there exists a route in $\mathcal{G}(\boldsymbol{\Omega}_{\boldsymbol{T}}, \phi)$ from the transition class that such a rule generates to the formula vertex, the rule is added to the rule base of the sliced agent (line 4). Finally, on line 7, $\mathsf{Slice}(P)$ is defined as the tuple containing the unaltered initial mental state and the new rule base. Thus, the slicing problem is solved by treating it as a reachability problem of the formula vertex in the reachability graph. In practice, we apply the PBS algorithm as follows. Given some property φ and program P, we first negate φ and convert it to NNF. Then, if $\neg\varphi \in \mathcal{L}_{\mathrm{LTL}}\setminus\{\mathsf{X},\mathsf{R}\}$, we construct the influence graph with respect to P and $\neg\varphi$. Finally, Alg. 3 is applied to P and the constructed graph. The actual verification procedure subsequently considers the sliced agent only.

The essence of the algorithm is line 3. Determining whether a route exists in the influence

---

1: $\overline{R} := \emptyset$
2: **for all** $\rho \in R$ **do**
3: **if** there exists a route $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi \in \mathit{Routes}(\mathcal{G}(\boldsymbol{\Omega}_{\boldsymbol{T}}, \phi))$ **then**
4: $\overline{R} := \overline{R} \cup \{\rho\}$
5: **end if**
6: **end for**
7: $\mathsf{Slice}(P) := \langle \mu_0, \overline{R} \rangle$

Algorithm 3: Let $P = \langle \mu_0, R \rangle$ be a GOAL agent, and let $\mathcal{G}(\boldsymbol{\Omega}_{\boldsymbol{T}}, \phi) = \langle \mathcal{V}, \mathcal{E} \rangle$. Then: this algorithm computes $\mathsf{Slice}(P)$.

graph can be done with well-known graph search algorithms such as depth-first search or breadth-first search. In both cases, we start at a vertex $\boldsymbol{\tau}(\rho)$, and explore the influence graph until either the formula vertex is reached (in which case a route $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi$ exists), or no more reachable yet unexplored vertices are left (in which case a route does not exist). Several optimisation schemes are possible. For example, if depth-first search is used, all transition classes that are on the search stack at the moment the formula vertex is reached also have a route to the formula vertex. Thus, these transition classes do not requires a separate search. Also, if a vertex $\boldsymbol{\tau}(\rho')$ is reached by a search (either depth-first or breadth-first) started from $\boldsymbol{\tau}(\rho)$, and we already know that there exists a route $\boldsymbol{\tau}(\rho') \rightsquigarrow \phi$, then there also exists a route $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi$ (because $\boldsymbol{\tau}(\rho) \rightsquigarrow \boldsymbol{\tau}(\rho') \rightsquigarrow \phi$). Similarly, if we already know that there does not exist a route from $\boldsymbol{\tau}(\rho')$ to the formula vertex, then continuing the search from $\boldsymbol{\tau}(\rho')$ to determine whether there exists a route from $\boldsymbol{\tau}(\rho)$ to the formula vertex (through $\boldsymbol{\tau}(\rho')$) is a waste of resources.

When is Alg. 3 correct? Well, at the beginning of Sect. 6.1, we briefly and informally discussed one condition that our algorithm must satisfy: the action rules of the agent that can influence the truth value of a negated property $\neg\varphi$ must be preserved. We can express this more formally as follows. Let P be a Goal agent, and let $\overline{P} = \mathsf{Slice}(P, \neg\varphi)$ be its slice with respect to $\neg\varphi$. Then, if P has a computation that satisfies $\neg\varphi$, $\overline{P}$ must also have a computation on which $\neg\varphi$ is satisfied. If this property is true, then $\overline{P}$ is called a *logically sound abstraction* of P [58]. Informally, if our slicing algorithm preserves logical soundness, then correctness of the sliced agent $\overline{P}$ always implies correctness of the original agent. That is, it excludes *false positives*. Logical soundness is, however, not sufficient: we also want our algorithm to construct $\overline{P}$ such that it is a *logically complete abstraction* of P [58]. This means that if $\overline{P}$ has a computation on which $\neg\varphi$ is satisfied, then P also has such a computation. As such, the possibility of *false negatives* is excluded.

The next theorem establishes that Alg. 3 preserves both logical soundness and completeness. It relies on three lemmas that we prove in Appxs. C.3.2,C.3.3. First, Lemma 13 establishes the existence of a minimal ϕ-satisfying computation as outlined in Sect. 6.2.3. Second, Lemma 14 states that if an agent has a computation with a finite prefix that is generated solely by transitions from the influential set, then the agent's slice has a computation with an equal prefix. The intuition here is that because all transitions in the influential set belong to transition classes that are retained in the slice, these transitions are also present in the transition system of the sliced agent. Consequently, all finite paths (e.g. prefixes of a computation) through the transition system of the original agent generated by only executing transitions in the influential set are also paths through the transition system of the sliced agent. The proof of Lemma 14 is by induction on the size of such a path, and shows that we can incrementally construct a path through the transition system of the sliced agent that is identical to the path through the transition system of the original agent. Finally, Lemma 15 establishes that each computation of a sliced agent is also a computation of the original agent; the proof is similar to that of Lemma 14.

In the proof of Theorem 13, the notion of *progression* reoccurs, which we briefly introduced at the end of Sect. 6.2.3: informally, $\mathit{progression}(\phi, i, \boldsymbol{\pi})$ denotes the LTL formula that must be true in $\boldsymbol{\pi}_i$ for the entire computation to satisfy ϕ; we refer to Appx. C.3.2 for the details of progression. The proof of Theorem 13 is straightforward application of previously introduced lemmas. To prove preservation of logical soundness ($\Rightarrow$), we show that if there exists a computation that satisfies an LTL$\backslash\{\mathsf{X},\mathsf{R}\}$ formula ϕ, then there exists a minimal ϕ-satisfying computation on which satisfaction of ϕ can be established on the prefix of size k (this is formalised with progression, i.e. in the k-th mental state on the computation, the formula that must be true is $\top$). Then, because the k-prefix of the minimal ϕ-satisfying computation contains transitions from $\mathsf{Influential}$ only, by Lemma 14, the sliced agent has a corresponding computation, whose k-prefix also satisfies ϕ. The proof in the opposite direction, i.e. to prove preservation of logical completeness ($\Leftarrow$), is very similar but relies on Lemma 15, and makes use of the fact that all transitions in the transition system of the sliced agent are influential by construction of the slice (i.e. if an action rule is retained in the slice, then the corresponding transition class is a subset of the influential set as stated by Theorem 12). In the following, recall that $\models_{\mathrm{LTL}}$ is LTL's entailment relation as defined in Sect. 3.1.1.

Theorem 13. *Let ϕ be an LTL$\backslash\{\mathsf{X},\mathsf{R}\}$ formula, let P be a Goal agent, let $\overline{P} = \mathsf{Slice}(P, \phi)$, let*

$\boldsymbol{\Pi}$ be the set of computations of P, and let $\overline{\boldsymbol{\Pi}}$ be the set of computations of $\overline{P}$. Then:

$$\begin{array}{l} \text{there exists a } \boldsymbol{\pi} \in \boldsymbol{\Pi} \\ \text{such that } \boldsymbol{\pi} \models_{\text{LTL}} \phi \end{array} \quad \Leftrightarrow \quad \begin{array}{l} \\ \text{there exists a } \overline{\boldsymbol{\pi}} \in \overline{\boldsymbol{\Pi}} \\ \text{such that } \overline{\boldsymbol{\pi}} \models_{\text{LTL}} \phi \end{array}$$

Proof.

($\Rightarrow$) If there exists a $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ such that $\boldsymbol{\pi} \models_{\text{LTL}} \phi$, then, by Lemma 13, there exist a $\boldsymbol{\pi}' \in \boldsymbol{\Pi}$ and $k \geq 0$ such that $progression(\phi, k, \boldsymbol{\pi}') = \top$, and for all $0 \leq i < k$, there exists a $\tau_i \in \mathsf{Influential}(\phi)$ such that $\langle \boldsymbol{\pi}_i, \boldsymbol{\pi}_{i+1} \rangle \in \tau_i$.
Then, by Lemma 14, there exists a $\overline{\boldsymbol{\pi}} \in \overline{\boldsymbol{\Pi}}$ such that $prefix(\boldsymbol{\pi}', k) = prefix(\overline{\boldsymbol{\pi}}, k)$.
Hence, $progression(\phi, k, \overline{\boldsymbol{\pi}}) = \top$.
Hence, $\overline{\boldsymbol{\pi}} \models_{\text{LTL}} \phi$.

($\Leftarrow$) Because for each transition class $\boldsymbol{\tau}(\rho)$ of $\overline{P}$ holds $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi \in Routes(\mathcal{G}(\boldsymbol{\Omega}_{\boldsymbol{T}}, \phi))$ by definition of Alg. 3, $\boldsymbol{\tau}(\rho) \subseteq \mathsf{Influential}(\phi)$ for each transition class $\boldsymbol{\tau}(\rho)$ of $\overline{P}$ by Theorem 12.
Hence, if there exists a $\overline{\boldsymbol{\pi}} \in \overline{\boldsymbol{\Pi}}$ such that $\overline{\boldsymbol{\pi}} \models_{\text{LTL}} \phi$, then there exists a $k \geq 0$ such that $progression(\phi, k, \overline{\boldsymbol{\pi}}) = \top$, and for all $0 \leq i < k$, there exists a $\tau_i \in \mathsf{Influential}(\phi)$ such that $\langle \overline{\boldsymbol{\pi}}_i, \overline{\boldsymbol{\pi}}_{i+1} \rangle \in \tau_i$.
Then, by Lemma 15, there exists a $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ such that $prefix(\overline{\boldsymbol{\pi}}, k) = prefix(\boldsymbol{\pi}, k)$.
Hence, $progression(\phi, k, \boldsymbol{\pi}) = \top$.
Hence, $\boldsymbol{\pi} \models_{\text{LTL}} \phi$.

This establishes the theorem. □

6.4 Case Studies

The interpreter-based GOAL model checker presented in Chap. 3 has been extended with the slicing algorithm presented in the previous section. More specifically, the GOAL-specific program component of the model checker has been extended with Alg. 3; all other components of the system did not incur any change. In this section, we treat three small case studies on the performance of the slicing algorithm. First, we revisit the motivating example of the chapter's introduction. Subsequently, we consider an agent based on the blocks world and counting domains that were under investigation in the performance comparison of Sect. 4.3. Third, we look at a more complex domain, namely the *wumpus world*. We conclude the section with a discussion on the obtained results.

6.4.1 `blenderAgent`

In the introduction of this chapter, we motivated slicing with an example in which `blenderAgent` is verified with respect to the property that it never believes to have put more bananas into the blender's reservoir than prescribed by the recipe. This property was formulated in LTL as $\varphi =$ `G bel(recipe(bananas,Qr),added(bananas,Q),Q=<Qr)`. Because the negation of this property is an LTL\{X,R} formula, namely $\neg\varphi =$ `F ¬bel(recipe(bananas,Qr),added(bananas,Q),Q=<Qr)`, our slicing algorithm is applicable.[1]

When we run the model checker with slicing switched on, a transition system with four mental states is generated and investigated (exactly the transition system shown in Fig. 6.1); the property

[1] It may be interesting to note that the implicit existential quantification that occurs when `recipe(bananas,Qr),added(bananas,Q),Q=<Qr` is fired as a PL query to the Prolog engine is turned into an implicit universal quantification by placing the ¬ operator in front of it (as happens when turning $\neg\varphi$ to NNF). That is, `¬bel(recipe(bananas,Qr),added(bananas,Q),Q=<Qr)` is only true if there does not exist a substitution that binds `Qr` and `Q` such that `recipe(bananas,Qr),added(bananas,Q),Q=<Qr` is derivable from the belief and knowledge bases, i.e. if *for all* possible bindings of `Qr` and `Q`, `recipe(bananas,Qr),added(bananas,Q),Q=<Qr` is not derivable from the belief and knowledge bases.

is reported true. The entire execution takes 498 milliseconds, and requires 3 MB of memory. Of these 498 milliseconds, 431 milliseconds were used to compute the slice. In the slice, two of the five action rules have been retained; the others are sliced away. Now, if we switch slicing off, the model checker explores the full transition system consisting of 18 mental states. This takes only 463 milliseconds and requires 6 MB of memory. Thus, although the slicing algorithm reduces the transition system substantially, namely by $1 - \frac{4}{18} = 78\%$, the entire model checking procedure takes more time to finish when slicing is enabled. That is, the overhead of computing slices is too high.

However, note that the size of the transition system is relatively small in the above example, while the costs of slicing are a one-time-expense. This might suggest that if the full transition system is larger, benefits may be gained because the costs of computing the slice become negligible with respect to resource consumption when verifying the full transition system. To test this hypothesis, we gave `blenderAgent` a larger blender and changed the recipe a bit. That is, rather than two bananas and two oranges, `blenderAgent` should now put i bananas and i oranges into the reservoir. Resource consumption of model checking with and without slicing for $i \in \{2, 4, 8, 16, 32, 64\}$ are shown in Table. 6.1.

The results in the table validate our hypothesis that when the transition system is larger, the overhead of running the slicing algorithm becomes negligible compared to the benefits. Note that the readings in the "Slice computation" row are roughly constant, i.e. they do not grow larger as the transition system increases. The reason is that the action rules of the agent do not change if i grows larger, i.e. the computations required to compute the slice are invariant to change of i. Thus, slicing can be very lucrative: for $i = 4$, the reduction of verification time is already 43%, growing to a 97% reduction in case $i = 64$. The measurements on memory consumption are only indicative like in Chap. 4 (see Sect. 4.2.3), but a definite (positive) trend is observable in this respect as well. A plausible explanation for this is that due to the reduction of the transition system, less states need be stored such that memory demands are lower.

	$i = 2$	$i = 4$	$i = 8$	$i = 16$	$i = 32$	$i = 64$
States						
Pbs enabled	4	6	10	18	34	66
Pbs disabled	18	38	102	326	1158	4358
Reduction	*78%*	*84%*	*90%*	*94%*	*97%*	*98%*
Verification time						
Pbs enabled	498 ms	521 ms	580 ms	718 ms	911 ms	1309 ms
• Slice computation	431 ms	424 ms	419 ms	443 ms	431 ms	425 ms
• Exploration	67 ms	97 ms	161 ms	275 ms	460 ms	884 ms
Pbs disabled	463 ms	917 ms	2116 ms	4849 ms	14068 ms	47887 ms
Reduction	*-7%*	*43%*	*73%*	*85%*	*94%*	*97%*
Memory consump.						
Pbs enabled	3 MB	6 MB	6 MB	6 MB	12 MB	12 MB
Pbs disabled	6 MB	12 MB	12 MB	23 MB	24 MB	23 MB
Reduction	*50%*	*50%*	*50%*	*74%*	*50%*	*48%*

Table 6.1: Resource consumption during verification of `blenderAgent` with respect to the property φ = `Gbel(recipe(bananas,Qr),added(bananas,Q),Q=<Qr)`, and in which i is the number of bananas and oranges that need be added to the blender's reservoir.

```
1  main: blocksCounterAgent{
2    beliefs{
3        block(aa). block(ab). block(ac). block(ad). block(ae).
4        on(aa,table). on(ab,aa). on(ac,ab). on(ad,ac). clear(ad).
5        on(ae,table). clear(ae).
6        current(0). target(4).
7    }
8    goals{
9      on(ab,table).
10   }
11   program{
12     if goal(on(ab,table)), bel(on(X,Y)), bel(clear(X))
13     then moveXfromYtoTable(X,Y).
14     %
15     if not(goal(current(I))), bel(current(J), not(target(J), K is J+1)
16     then adopt(current(K)).
17     %
18     if a-goal(current(K)), bel(J is K-1)
19     then updateCurrent(J,K).
20   }
21   actionspec{
22     moveXfromYtoTable(X,Y){
23       pre{ block(X) }
24       post{ not(on(X,Y)), on(X,table), clear(Y) }
25     }
26     updateCurrent(J,K){
27       pre{ true }
28       post{ not(current(J)) , current(K) }
29     }
30   }
31 }
```

Figure 6.3: Merged blocks world / counter agent, i.e. `blocksCounterAgent`. (Blocks are labelled `aa`, `ab`, et cetera rather than `a`, `b`, et cetera for "historical reasons": in the original experiments featuring `blocksAgent`, presented in Sect. 4.3, more than 26 blocks were present such that single-letter labels were inadequate. For ease of reference, we chose to use the same labels here.)

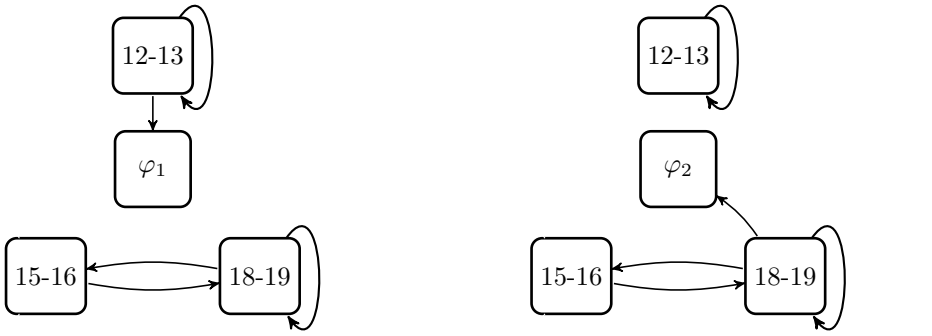
6.4.2 Counting Blocks

In this case study, the agents of Experiment 1 (blocks world; see Sect. 4.3.1) and Experiment 3 (counting without memorising; see Sect. 4.3.3) in Chap. 4 are merged into one agent, called `blocksCounterAgent`. The code by which `blocksCounterAgent` is defined is given in Fig. 6.3; the only difference with the two individual agents is that `blocksCounterAgent` has an upper bound on the number to which it can count, by means of the `target/1` predicate. This is to ensure that the state space of the transition system is finite; as a consequence, we can check properties whose truth value can only be established by exploring the entire transition system (in contrast to the properties under investigation in Sect. 4.3.3, whose truth could be established before the entire transition system was generated). The simplicity of this case study's set-up enables us to illustrate the use of a rule of thumb for determining whether slicing may lead to a reduction of the transition system in Sect. 6.4.4. Such a rule of thumb is needed, because slicing does not always lead to a reduction: if all transition classes have a route to the formula vertex in the influence graph, all action rules are retained such that no reduction is gained. Identifying when and when not the formula vertex is reachable by all transition classes is hence important.

To use the slicing algorithm, the negation of the property in NNF must be in LTL$\setminus\{$X,R$\}$. The following two properties satisfy this constraint:

$$\begin{aligned} \varphi_1 &= \mathsf{G}[\mathbf{bel}(\texttt{on(ab,table)}) \rightarrow \neg\mathbf{bel}(\texttt{on(X,Y),not(Y=table)})] \\ \varphi_2 &= \mathsf{G}\,\mathbf{bel}(\texttt{current(K),target(T),not(K>T)}) \end{aligned}$$

The first property, φ_1, formalises that if the agent's initial goal is achieved (having `ab` on the table), indeed all blocks are on the table (this was already informally argued for in Sect. 4.3.1). The second property, φ_2, states that the agent will never count beyond its upper bound. For the verification of φ_1, the model checker will search for a computation on which $\neg\varphi_1 = \mathsf{F}[\mathbf{bel}(\texttt{on(ab,table)}) \wedge \mathbf{bel}(\texttt{on(X,Y),not(Y=table)})]$ is true. Similarly, to verify φ_2, the model checker searches for



(a) $\varphi_1 = \mathsf{G}[\mathbf{bel}(\texttt{on(ab,table)}) \rightarrow \neg\mathbf{bel}(\texttt{on(X,Y),not(Y=table)})]$

(b) $\varphi_2 = \mathsf{G}\,\mathbf{bel}(\texttt{current(K),target(T),not(K>T)})$

Figure 6.4: Influence graphs for `blocksCounterAgent` with respect to φ_1 and φ_2.

a computation that satisfies $\neg\varphi_2 = \mathsf{F}\neg\mathbf{bel}(\texttt{current(K),target(T),not(K>T)})$. The influence graphs for `blocksCounterAgent` with respect to φ_1 and φ_2 are given in Fig. 6.4. The labels on vertices (recall that vertices are transition classes) correspond to the line numbers on which the action rule of the corresponding transition class is defined.

When we slice `blocksCounterAgent` with respect to φ_1, the slice contains only one of the three action rules (namely the rule on lines 12-13 in Fig. 6.3); when slicing with respect to φ_2, two action rules are retained (namely those on lines 15-19). In fact, the Pbs algorithm transforms the merged `blocksCounterAgent` back to the two agents from which it was composed. Nevertheless, the reductions, given in Table 6.2 for various values of i in `target(`i`)`, are substantial. Even when the full transition system is only small, i.e. consists of 36 states, the benefits of Pbs outweigh the overhead of running the algorithm. As in the previous subsection, these benefits grow larger as the full transition systems grows larger. Thus, slicing is again very much worth the effort.

We acknowledge that the practical use of only merging two independent agents with each other is not directly obvious. Therefore, to make things a little more interesting, we add dependencies

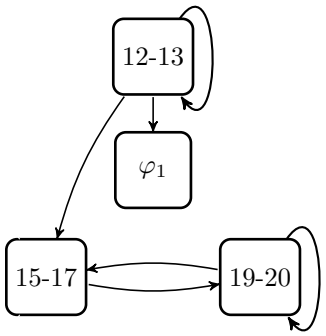
	$\varphi_1, i=4$	$\varphi_1, i=16$	$\varphi_1, i=64$	$\varphi_2, i=4$	$\varphi_2, i=16$	$\varphi_2, i=64$
States						
Pbs enabled	4	4	4	9	33	129
Pbs disabled	36	132	516	36	132	516
Reduction	*89%*	*97%*	*99%*	*75%*	*75%*	*75%*
Verification time						
Pbs enabled	275 ms	284 ms	290 ms	399 ms	807 ms	1907 ms
• Slice computation	168 ms	178 ms	175 ms	183 ms	183 ms	177 ms
• Exploration	107 ms	106 ms	115 ms	216 ms	624 ms	1730 ms
Pbs disabled	837 ms	2218 ms	6008 ms	886 ms	2324 ms	5981 ms
Reduction	*67%*	*87%*	*95%*	*55%*	*65%*	*68%*
Memory consump.						
Pbs enabled	3 MB	3 MB	3 MB	3 MB	6 MB	12 MB
Pbs disabled	12 MB	13 MB	25 MB	10 MB	12 MB	25 MB
Reduction	*25%*	*77%*	*88%*	*30%*	*50%*	*52%*

Table 6.2: Resource consumption during verification of `blocksCounterAgent` with respect to the properties $\varphi_1 = \mathsf{G}[\mathbf{bel}(\texttt{on(ab,table)}) \rightarrow \neg\mathbf{bel}(\texttt{on(X,Y),not(Y=table)})]$ and $\varphi_2 = \mathsf{G}\,\mathbf{bel}(\texttt{current(K),target(T),not(K>T)})$.

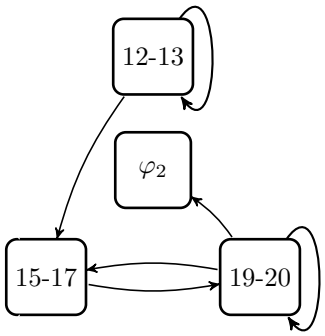
```
15    if not(goal(current(I))), bel(current(J), not(target(J)), K is J+1),
16      bel(on(X, table), not(counted(X)))
17    then adopt(current(K),counted(X)).
18    %
19    if a-goal(current(K), counted(X)), bel(J is K-1)
20    then updateCurrent(J,K,X).
```

```
26    updateCurrent(J,K,X){
27      pre{ true }
28      post{ not(current(J)), current(K), counted(X) }
29    }
```

Figure 6.5: Changes required to turn `blocksCounterAgent` into `blocksCounterAgent′`.



(a) $\varphi_1 = \mathsf{G}[\textbf{bel}(\texttt{on(ab,table)}) \rightarrow \neg\textbf{bel}(\texttt{on(X,Y),not(Y=table)})]$



(b) $\varphi_2 = \mathsf{G}\,\textbf{bel}(\texttt{current(K),target(T),not(K>T)})$

Figure 6.6: Influence graphs for `blocksCounterAgent′` with respect to φ_1 and φ_2.

between the agent's action rules. Specifically, we replace lines 15-19 and lines 26-29 in Fig. 6.3 by the code in Fig. 6.5. We call the resulting agent `blocksCounterAgent′`: due to the changes, `blocksCounterAgent′` counts the blocks that are on the table, and each block only once. At least, that is what we aimed at. To verify that our changes indeed bring about their intended effect, we first verify the following property:

$$\varphi \quad = \quad \mathsf{FG}\,\textbf{bel}(\texttt{current(5)}) \wedge \mathsf{G}\neg\textbf{goal-a}(\texttt{counted(X)})$$

This property expresses that eventually, `blocksCounterAgent` always believes to have counted to 5, and that it never has the goal to count a block that has already been counted. Provided that there are five blocks, this expresses that each block is counted only once. Fortunately, the model checker reports that the property is satisfied. Note that we cannot use our slicing algorithm when verifying φ because its negation in NNF is not an LTL$\backslash\{\mathsf{X},\mathsf{R}\}$ formula.

Figure 6.6 shows the influence graphs for the `blocksCounterAgent′` with respect to φ_1 and φ_2; the differences with Fig. 6.4 are well observable. When slicing `blocksCounterAgent′` with respect to φ_1, we obtain exactly the same slice as for `blocksCounterAgent` in Fig. 6.3. This is not surprising since neither the action rule that can make the `moveXfromYtoTable` action an option, nor this action's precondition have references to the newly introduced `counted/1` predicate. In contrast, the slice with respect to φ_2 does not yield a reduction any more: all action rules can influence φ_2's truth or falsehood. The reason is that the counting process is influenced by the configuration of the blocks (i.e. whether a block is already on the table or not) such that performance of `moveXfromYtoTable`, which alters this configuration, might generate influential transitions. Thus, slicing the altered `blocksCounterAgent` with respect to φ_2 does not reduce the transition system.

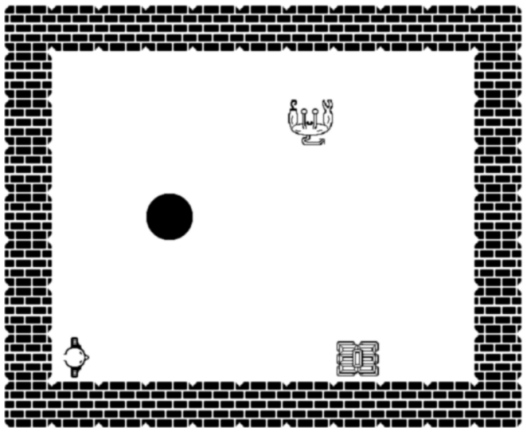


Figure 6.7: Example of a wumpus world. The agent is in the bottom left corner, the heap of gold is some steps in front of him, the only pit is depicted by the black circle, and the wumpus is the remaining creature. The cave is surrounded by walls.

6.4.3 Wumpus World

Finally, in this last case study, we consider a scenario known as the *wumpus world* to show that slicing does not always work. Specifically, if an agent needs to carry out subtasks that are influenced by other subtasks, slicing will not be effective.

In [85], the wumpus world is described as follows.

> The wumpus world is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only mitigating feature of living in this environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it makes an excellent testbed environment for intelligent agents. Michael Genesereth was the first to suggest this.

An example of a simple wumpus world is given in Fig. 6.7.

There exist many solutions to the wumpus world problem; we had access to a repository containing seven different solutions written in Goal. All these agents were designed by groups of students for an M.Sc. course on artificial intelligence; they operate at different levels of performance, where performance is measured as the number of actions it takes the agent to obtain the heap of gold (if possible at all), and escape the cave in several different cave instances. Though there are ample differences between these seven agents, they have in common that their initial belief bases contain `hasarrow`, expressing that the agent still has its only arrow, and `wumpusisalive`, expressing that the wumpus is alive. Similarly, their initial goal bases contain the single goal $\{\texttt{hasgold}, \texttt{getoutofthiscave}\}$, expressing that the agent wants to obtain the gold and escape.

Interesting properties that these agents might satisfy are the following:

$$\begin{aligned} \varphi_1 &= \mathsf{G}[\mathbf{bel}(\texttt{hasarrow}) \vee \neg\mathbf{bel}(\texttt{wumpusisalive})] \\ \varphi_2 &= \mathsf{G}[[\neg\mathbf{goal}(\texttt{getoutofthiscave})] \rightarrow \mathbf{bel}(\texttt{hasgold})] \end{aligned}$$

The first property states that either the agent believes that it still has its arrow, or it believes that the wumpus is no longer alive. Assuming that the wumpus can only die if it is shot by the agent, this property expresses that if the agents shoots it never misses (we acknowledge that it is debatable whether this really is a desirable property, but that is beyond the scope of this discussion). The second property states that if the agent does not have the goal to leave the cave,

then it has acquired the heap of gold. Because **goal**(`getoutofthiscave`) is true in the initial mental state, for ¬**goal**(`getoutofthiscave`) to be true, the agent has either dropped this goal explicitly, or achieved it by climbing out of the cave.

Unfortunately, slicing the seven agents with respect to φ_1 and φ_2 does not result in the removal of any action rules (except for action rules whose consequent is a **skip** action, and those that make the agent climb out of the cave). In other words, no reduction of the transition system is gained for any of the seven agents (we have not run the model checker afterwards). This is surprising because the properties under investigation seem to address subtasks of the agent that do not influence its other subtasks:

- With respect to φ_1, one might believe that the transitions generated by action rules for grabbing the gold do not influence transitions generated by rules that implement the agent's aiming and shooting capabilities. The reason that this is not the case is as follows. Grabbing the gold makes **bel**(`hasgold`) true. In all implemented agents, this influences the enabledness of transitions that are generated by action rules that deal with escaping the cave. During cave escaping, the position and orientation of the agent are altered. As a consequence (the agent's position and orientation are always of relevance for shooting the wumpus), grabbing the gold might indirectly influence the agent's shooting behaviour.

- With respect to φ_2, whether **bel**(`hasgold`) is true can depend on whether the wumpus is dead or alive: the heap of gold may be on the same location as the wumpus, or the wumpus may block the way to the heap. Consequently, transitions that are generated by action rules responsible for killing the wumpus can indirectly influence the truth or falsehood of φ_2.

6.4.4 Discussion

Although the results of the first two case studies were very promising, the third case study shows that slicing has limitations. Specifically, the wumpus world example demonstrates that subtasks of which one might think they do not influence each other can nevertheless be linked, prevent the slicing algorithm from reducing the transition system significantly, if any reduction is gained at all.

An important question that now arises is how often PBS will be able to yield a significant reduction of the transition system in practice. By lack of more experimental data for GOAL, we look at the effectiveness of slicing in the field of imperative language model checking. In a 2006 investigation on the possible merits of PBS in [35], it is concluded that although many researchers have developed negative opinions about slicing, it is a technique that is efficient to apply and yields non-trivial reductions of verification time. This conclusion is based on experimental data obtained when slicing and verifying JAVA programs. However, in [35], the scope is limited to verification of deadlocks and assertions; LTL properties are not under investigation. Also, Holzmann, lead designer of SPIN, has indicated that SPIN's built-in slicing algorithm is not effective for real-world PROMELA models [35]. As such, there seems to be a gap between theory and practice with respect to slicing methods (in model checking): they are called effective in papers presenting them (e.g. [14, 22, 75, 102]), but do not live up to the expectations when applied to real-world problems.

Based on these observations, we believe that it a necessity to have some rule of thumb for predicting whether the slicing algorithm might be able to yield a reduction for a specific case. We formulate such a rule of thumb informally as follows: *if the agent needs to carry out a subtask that influences neither other subtasks nor the property under investigation, then slicing may be advantageous.* To use this rule of thumb, no influence graphs need be constructed; it can be applied by merely having knowledge about what the agent is *supposed* to do. We emphasise "supposed", because there is a danger here: subtasks that *should* be independent may not be so due to an error in the implementation. Consequently, based on the rule of thumb, one might believe that slicing will be beneficial while it actually is not because of such a bug. Another potential drawback of the rule of thumb is that subtasks that are deemed independent may not actually be so. We ourselves made this error in judgement in the wumpus world example. Nevertheless,

the above rule of thumb can serve as a guideline and as such can be helpful to determine whether slicing should be applied or not.

We now informally argue how this rule of thumb corresponds to the influence graph based on its *strongly connected components* (Scc). In graph theory (e.g. [3]), an Scc of a graph $\mathcal{G}$ is a sub-graph of $\mathcal{G}$, whose every vertex is reachable by a path from all other vertices in the same sub-graph. Next, the *condensation* of $\mathcal{G}$ is a graph in which all Sccs are contracted to a single vertex. Subsequently, we rephrase the rule of thumb as follows: *if the condensation of the influence graph of an agent with respect to a property contains at least two vertices without outgoing edges, then slicing may be advantageous.* One of these vertices without outgoing edges is always the formula vertex, as this vertex has no outgoing edges by definition. The idea is that if a second such vertex exists in the condensation, then there exists an Scc $\mathcal{G}'$ in the original influence graph whose all vertices neither have an outgoing edge to the formula vertex, nor to another Scc $\mathcal{G}''$. Each Scc can be viewed as a subtask of an agent, and the Sccs $\mathcal{G}'$, whose corresponding vertex in the condensation has no outgoing edges, are subtasks that do not influence other subtasks.

We exemplify this based on Figs. 6.4,6.6. The influence graph in Fig. 6.4a has three Sccs: $\{12\text{-}13\}$, $\{\varphi_1\}$, and $\{15\text{-}16, 18\text{-}19\}$: the first Scc corresponds to the tower deconstruction subtask, the second Scc corresponds to the formula, and the third Scc corresponds to the the counting subtask. Because we know that the tower deconstruction subtask does not influence the counting subtask (and vice versa), the original rule of thumb tells us that slicing may be beneficial. Likewise, because there exist two Sccs whose corresponding vertex in the condensation has no outgoing edges (namely $\{\varphi_1\}$, and $\{15\text{-}16, 18\text{-}19\}$), the reformulated rule of thumb tells the same. Indeed, running the Pbs algorithm was beneficial in this case, as demonstrated by Table 6.2. To the influence graphs in Figs. 6.4b,6.6a, similar reasoning applies. For Fig. 6.6b, however, things are slightly different. Although the Sccs are still the same, there is only one vertex without outgoing edges in the condensation (namely Scc $\{\varphi_2\}$) instead of two. Consequently, our reformulated rule of thumb does not suggest that slicing may be beneficial. The original rule of thumb says the same, as we know that the tower deconstruction subtask influences the counting subtask, and that the counting task influences the property. Indeed, slicing `blocksCounterAgent'` with respect to φ_2 did not yield a reduction.

We end this section with the remark that slicing can be used for many more applications other than model checking. An overview is given in [95], and includes program analysis, program differencing (the process of comparing an old and a new version of a program to see which parts have changed and as such require testing), software maintenance, testing, and compiler / interpreter tuning. Although we have not investigated these applications of slicing Goal agents as it is beyond the scope of this thesis, our slicing algorithm may be compatible (perhaps with minor modifications) with previously mentioned applications as well. As such, the algorithm is relevant not only to Goal model checking, but to Goal software engineering in general.

6.5 Extensions

In this section, we discuss several extensions to the slicing algorithm. The correctness of the algorithm has not been proven with respect to the extensions to be proposed; they should be regarded as conjectures, and can be seen as possible topics for future research on slicing Goal agents.

6.5.1 Strong Next

Throughout the chapter, we have considered the {X,R}-free fragment of LTL. This restriction can be lifted partially by introducing the *strong next* operator X_S. Informally, the only difference between X and X_S is that X_S imposes an additional constraint on its argument, namely that it must be false in the current mental state. More formally, the semantics of X_S are defined as follows (the semantics of $\neg$, $\wedge$, and X are repeated from Sect. 3.1.1). Let $\boldsymbol{\pi}$ be a computation, and let i be a time point.

$$
\begin{array}{lll}
\boldsymbol{\pi}, i \models_{\text{LTL}} \neg\phi & \text{iff} & \boldsymbol{\pi}, i \not\models_{\text{LTL}} \phi \\
\boldsymbol{\pi}, i \models_{\text{LTL}} \phi \wedge \phi' & \text{iff} & \boldsymbol{\pi}, i \models_{\text{LTL}} \phi \text{ and } \boldsymbol{\pi}, i \models \phi' \\
\boldsymbol{\pi}, i \models_{\text{LTL}} \mathsf{X}\phi & \text{iff} & \boldsymbol{\pi}, i+1 \models_{\text{LTL}} \phi \\
\boldsymbol{\pi}, i \models_{\text{LTL}} \mathsf{X}_S\phi & \text{iff} & \boldsymbol{\pi}, i+1 \models_{\text{LTL}} \phi \text{ and } \boldsymbol{\pi}, i \not\models \phi
\end{array}
$$

Thus, $\mathsf{X}_S\phi$ is an abbreviation for $[\mathsf{X}\phi] \wedge \neg\phi$.

Now, if $\mathsf{X}_S\phi$ is true in some mental state μ on $\boldsymbol{\pi}$, then the transition τ executed in μ *must* alter the truth value of ϕ. As such, τ is in the influential set with respect to $\mathsf{X}_S\phi$. Consequently, the issue that caused us to restrict the use of X (namely the observation that an X formula can be true without execution of a transition in the influential set) is resolved (see also Sect. 6.2.3). Unfortunately, practical use of X_S might be limited. To see this, first note that the X_S operator must occur in the *negated* property $\neg\varphi$, and not in the property φ itself. This means that X formulas are only allowed in φ, if the NNF of φ contains sub-formulas of the form $\varphi' = [\mathsf{X}\phi] \rightarrow \phi$ due to the following derivation:

$$
\begin{array}{rcll}
\mathsf{X}_S\phi & = & [\mathsf{X}\phi] \wedge \neg\phi & \\
& = & [\mathsf{X}\neg\neg\phi] \wedge \neg\phi & \\
& = & [\neg\neg\mathsf{X}\phi] \wedge \neg\phi & \\
& = & \neg[[\neg\mathsf{X}\phi] \vee \phi] & \\
& = & \neg[[\mathsf{X}\phi] \rightarrow \phi] & = \quad \neg\varphi' \text{ in which } \varphi' = [\mathsf{X}\phi] \rightarrow \phi
\end{array}
$$

To us, it is not directly obvious whether the addition of $[\mathsf{X}\phi] \rightarrow \phi$ to the LTL fragment that can be used in combination with the slicing algorithm increases expressiveness significantly. For example, $\mathsf{G}[[\mathsf{X}\phi] \rightarrow \phi]$ can also be specified without X, namely as $[\mathsf{G}\phi] \vee \mathsf{G}\neg\phi$.

6.5.2 Dynamic Slicing

With *dynamic slicing* of imperative languages (see, for example, [95]), certain assumption about the input of the program are made when the slice is constructed, e.g. that system variable x is 0 and y is 1 at the start of execution. In contrast, with *static slicing* (of which the algorithm presented in this chapter is an instance), no assumptions about the input are made. When applied to GOAL agents, a dynamic slicing algorithm would also take the initial mental state into account.

One straightforward way to do this is by slightly altering the definition of a route. That is, in our current definition, every transition class in the influence graph can be the start of a route. When also taking the initial mental state μ_0 into consideration, we can restrict the transition classes that may be the start of a route to those transition classes whose corresponding action rule generates an option in μ_0. That is, a route in such a case is a sequence (and all such sequence postfixes) of transition classes succeeded by the formula vertex such that the first transition class generates an option in μ_0. The intuition of this is that if all transitions in a transition class $\boldsymbol{\tau}(\rho)$ are not enabled in the initial mental state and will not become enabled by execution of a transition in any other transition class, then the transitions in $\boldsymbol{\tau}(\rho)$ (i.e. the transitions generated by ρ) can never influence the truth value of the negated property, simply because they are never enabled.

One argument in disfavour of dynamic slicing as proposed here is that is does not yield any additional reduction of the transition system as compared to static slicing. To see this, note that the method as proposed above only removes transitions that would never become enabled anyway. In other words, if the corresponding action rules would be retained (as the static slicing algorithm does), they would have no effect on the computations of an agent. However, although the transition system is not reduced, there may be benefits in terms of resource consumption during successor generation. That is, with less action rules present, less action rules (and, as such, mental state conditions) need be evaluated by the interpreter during action option generation. This can yield an additional reduction of verification time.

6.5.3 On-The-Fly Slicing

In the previous, a single slice was obtained for the whole program before commencing the model checking procedure. Benefits can be gained, however, when slices are generated during the generation and exploration (we assume an on-the-fly model checking regime) of the state space. We illustrate this informally with the following example.

Consider the property $\varphi = \mathsf{G}[\mathbf{bel}(\texttt{foo}) \rightarrow \mathsf{G}\,\mathbf{bel}(\texttt{bar})]$ whose negation in Nnf is $\neg\varphi = \mathsf{F}[\mathbf{bel}(\texttt{foo}) \wedge \mathsf{F}\neg\mathbf{bel}(\texttt{bar})]$. The agent under investigation is sliced with respect to $\neg\varphi$. Then, as soon as $\mathbf{bel}(\texttt{foo})$ is true in some mental state μ, the model checker will search for a path starting in μ on which $\neg\varphi' = \mathsf{F}\neg\mathbf{bel}(\texttt{bar})$ is true (if such path does not exist, the model checker continues in μ with the whole $\neg\varphi$). Because the mental state conditions that occur in $\neg\varphi'$ are a subset of the mental state conditions that occur in $\neg\varphi$, the slice obtained for $\neg\varphi$ may contain action rules that generate transition that cannot influence the truth value of $\mathbf{bel}(\texttt{bar})$ (i.e. they only influence the truth value of $\mathbf{bel}(\texttt{foo})$). That is, the slice is larger than necessary for establishing satisfaction of the specific (sub-)formula. Consequently, the number of paths that start in μ can be further reduced if the program is sliced with respect to the (sub-)formula that must be true in that particular mental state.

To minimise the overhead of computing slices on-the-fly during the combined generation and exploration of the transition system, at least the influence graph could be computed beforehand. Then, during actual verification, only the paths through the influence graph need be determined without expensive write set computation. Another possibility is to already pre-compute entire slices for each mental state condition that occurs in the property. Then, during actual verification, the slices corresponding to the Mscs that occur in the (sub-)formula that must be true in a specific mental state need only be merged.

We are not aware of on-the-fly slicing methods as proposed here in the literature. It has, however, been pointed out (e.g. in [23]) that smaller properties allow for greater reduction than larger properties in general. As such, it seems always advisable to verify as small a property as possible. For example, rather than verifying $\mathsf{F}\psi \wedge \mathsf{F}\psi'$, it is better to model check $\mathsf{F}\psi$ and $\mathsf{F}\psi'$ separately.

6.6 Summary

In this chapter, we have investigated property-based slicing for Goal agents. After outlining the idea of Pbs (with respect to model checking) in the first section, we have treated a graph-theoretic and relational account of how transitions influence each other and the property under investigation. The two formalisms were proven to correspond with each other, enabling the formulation of the algorithm in terms of the influence graph, while proving this algorithm correct using the influence relation / influential set. The algorithm was given in Sect. 6.3, along with its proof of correctness, which showed that the algorithm preserves both logical soundness and completeness. Preliminary case studies with an implementation of the algorithm show that slicing has the potential to yield substantial reductions, specifically in case the agent under verification has subtasks that neither influence each other, nor the property; we have argued that such subtasks correspond to Sccs in the influence graph. Finally, some extensions to the basic algorithm were proposed, which have not yet been implemented.

Chapter 7

Partial Order Reduction

In the previous chapter, we have discussed property-based slicing as a technique for reducing the transition system. This chapter presents another such technique, namely a *partial order reduction* (POR) algorithm for GOAL agents. In model checking, POR (or *model checking using representatives*) is a well known technique for reducing the size of the state space of the system under investigation. To the best of our knowledge, POR methods have up to now only been used for the verification of concurrent systems. In that context, POR algorithms try to exploit the observation that the various orders in which concurrent events can take place is often meaningless for the verification of a property. Nevertheless, a model checker without a POR algorithm needs to investigate all of them. The idea is to choose one representative order of concurrent events, and disregard all the others; applied to concurrent systems, POR methods can yield a reduction that is exponential in the degree of concurrency, i.e. the number of different concurrent processes [24].

The GOAL systems that we consider in this thesis are, however, not concurrent. Consequently, there are no concurrent events such that one might wonder how we can benefit from POR algorithms. The answer is that concurrency is not the only possible cause of different orders of the same events. In GOAL, an agent may non-deterministically choose to perform a sequence of actions, whose precise order is irrelevant. For example, whether `blenderAgent` first puts a banana into the blender followed by an orange, or first an orange followed by a banana does not matter, provided that the property under investigation does not distinguish between these orders. One such property is that `blenderAgent` eventually believes to have filled the blender, i.e. $\varphi = \mathsf{F}\,\mathbf{bel}(\texttt{filled})$. In this case, it is sufficient to investigate only one *representative* computation instead of all 16. Such a computation is shown in Fig. 7.1. Note that the negation of φ in NNF, i.e. $\neg\varphi = \mathsf{G}\neg\mathbf{bel}(\texttt{filled})$, is *not* in LTL$\backslash\{\mathsf{X},\mathsf{R}\}$ such that slicing is not possible in this case. An example of a property in which the order does play a role is that `blenderAgent` always first puts all bananas into the reservoir before any orange.

The observation that the different possible orders in which transitions can be executed, caused by GOAL's non-deterministic action selection, may be irrelevant, has been the primary motivation for the work in this chapter. The remainder is organised as follows. In Sect. 7.1, we discuss existing partial order reduction approaches and algorithms. Specifically, we have a look at the *ample set method*, which is the method that we will apply. Its GOAL-specific implementation is described in Sect. 7.2. In Section 7.3, we discuss case studies with the POR algorithm's implementation. Section 7.4 treats the complementarity of POR, and Sect. 7.5 concludes this chapter.

7.1 Theory

In this section, we discuss the theory behind POR. Specifically, we treat the *ample set method* (e.g. [79, 80]), as this is the method on which our POR implementation for GOAL agents is based (we motivate our choice for this method in Sect. 7.1.2). The ample set method is a member of a larger class of POR algorithms called *persistent set methods* (this name was coined in [46]). After a treatise that is specific to the ample set method in Sect. 7.1.1, we briefly discuss alternative

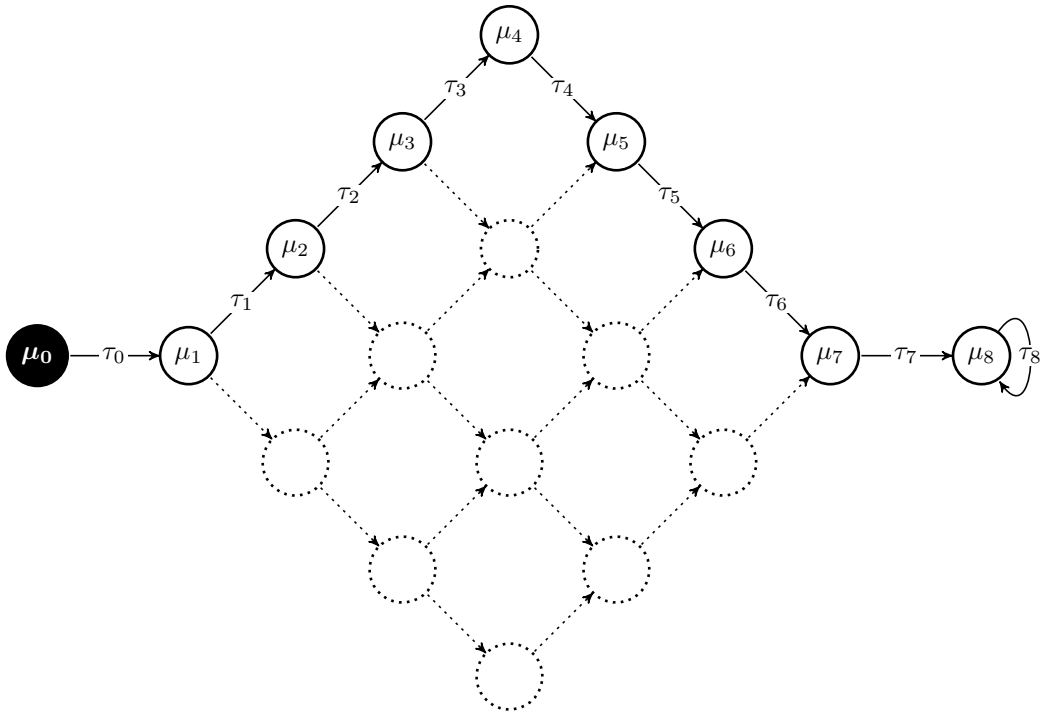


Figure 7.1: Ideal transition system of `blenderAgent` when verifying the property φ = `Fbel(filled)`.

persistent set methods, a different Por method that is compatible with persistent set methods, and Por efforts in the agent verification literature in Sect. 7.1.2. This section does not introduce new theory, and serves as an overview of past research.

The idea of Por methods in general (an extensive account is given in [46, 24]) is to construct a reduced transition system that is smaller than the full transition system. For simplicity of the exposition that follows, we assume a not-on-the-fly model checking regime in which we first construct the reduced transition system, and then apply an exploration algorithm to it. Applying the theory in an on-the-fly approach requires only minor changes [60]; we discuss these in Sect. 7.2.3. A desired characteristic of all Por methods is that they should be able to extract a reduced transition system from the full transition system without the need for complete generation of the latter. Persistent set methods try to achieve this by selecting a subset, called a *persistent set*, of the total set of enabled transitions in the state at that moment under investigation. Subsequently, only the transitions in the persistent set are executed, and the resulting successor states explored; the transitions that are also enabled but not in the persistent set are disregarded. By doing so, one hopes to reduce the number of reachable states.

In [24], three properties that persistent sets should satisfy are identified:

1. When a persistent set is used instead of the set of all enabled transitions, sufficiently many behaviours must be present in the reduced transition system such that the model checking algorithm gives correct results (i.e. the same results as when the full transition system would have been verified).

2. Using a persistent set should result in a significantly smaller transition system.

3. The overhead in computing a persistent set must be reasonably small.

The problem we are facing is, thus, how to compute persistent sets such that these three properties are satisfied. A naive solution is to only regard the independence relation between transitions (the

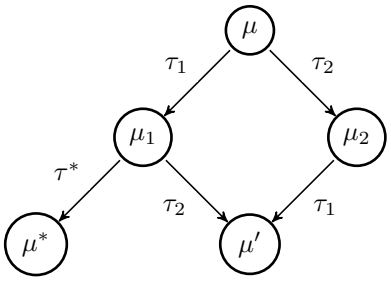


Figure 7.2: Simple transition system.

independence relation was introduced in Sect. 5.4.3). That is, one might be tempted to believe that only one order of two independent transitions τ_1 and τ_2 need be investigated by the model checker, as independent transitions satisfy the commutativity condition (see Sect. 5.4.3): it does not matter whether τ_1 is executed before τ_2 or vice versa in order to reach their common successor. This is, unfortunately, incorrect for the following two reasons [24]:

- *The property under investigation φ might be sensitive to the order in which the transitions are executed.*

  Consider, for example, the transition system in Fig. 7.2 in which transitions τ_1 and τ_2 are independent. Suppose the current state is μ, that the belief base in μ_1 equals $\{\texttt{foo}\}$, that the belief base in μ_2 equals $\{\texttt{bar}\}$, and that the belief base in μ' equals $\{\texttt{foo}, \texttt{bar}\}$ (this can happen for instance if the belief base in μ is empty, τ_1 adds `foo` to the belief base, and τ_2 adds `bar`). If the negated property is $\varphi = \mathsf{F}[\mathbf{bel}(\texttt{foo}) \wedge [\neg \mathbf{bel}(\texttt{bar})] \wedge \mathsf{F}\,\mathbf{bel}(\texttt{bar})]$, then choosing the path $\tau_1\tau_2$ as a representative and disregarding the path $\tau_2\tau_1$ causes the model checker to miss a violating behaviour, even though τ_1 and τ_2 are independent.

- *The intermediate states (i.e. the states resulting from the execution of τ_1 or τ_2) may have successors other than their common successor, which may not be explored if either one is disregarded.*

  To exemplify this, we reconsider the transition system in Fig. 7.2. Besides τ_2, τ^* is also enabled in μ_1: the execution of τ^* results in μ^*. Consequently, choosing the path $\tau_2\tau_1$ as a representative and disregarding the path $\tau_1\tau_2$ causes the model checker to miss the possibility of exploring μ^* (because μ_1 is never visited).

The first reason suggests that we should incorporate the notion of visibility during the computation of persistent sets (the visibility relation was introduced in Sect. 5.4.1). That is, if two transitions are *not* visible to φ, then φ is insensitive to the order in which these transitions are executed, because if transitions are invisible to φ, they cannot affect φ's truth value. The incorporation of visibility is, however, not yet sufficient, because a certain subset of LTL formulas are *always* sensitive to the order in which transitions are executed. Consequently, when such properties are under investigation, no POR algorithm can ever yield a reduction. Therefore, POR methods restrict themselves to those LTL formulas that do allow for the exploitation of order effects, namely the *stuttering invariant* LTL formulas (e.g. [24]). An LTL formula ϕ is stuttering invariant if for all *stuttering equivalent* computations $\boldsymbol{\pi}$ and $\boldsymbol{\pi}'$, $\boldsymbol{\pi} \models \phi$ iff $\boldsymbol{\pi}' \models \phi$. Informally, two computations $\boldsymbol{\pi}$ and $\boldsymbol{\pi}'$ are stuttering equivalent if replacing every finite sequence of the same mental state with a single occurrence of this mental state in both computations results in two equal computations [80]. It can be shown that the formulas in the X-free fragment of LTL, denoted LTL$\backslash\{\mathsf{X}\}$, are stuttering invariant [68]. Moreover, in [81], it has been proven that *all* stuttering invariant LTL formulas are in LTL$\backslash\{\mathsf{X}\}$. In the remainder of this chapter, we only consider formulas that are in LTL$\backslash\{\mathsf{X}\}$.

The second reason suggests that we need a means to somehow predict in the current mental state how the (restricted) exploration of direct successor mental states will affect the mental states

that will be reached during the entire exploration phase. As the various persistent set methods approach this problem from different angles, we now have a closer look at the ample set method (up to now, the treatise was generic to all persistent set methods). For convenience and future reference, we use Goal terminology and notation in parts of the exposition that are inherent to the method, and not specific to its application to Goal.

7.1.1 Ample Set Method

The ample set method (see [24] for a detailed account), like other persistent set methods, was introduced in the early 1990s, when Por techniques for model checking were emerging. It does not define a specific algorithm for establishing which transitions can be disregarded, but specifies four conditions (listed in Table 7.1, and treated shortly) that a set of transitions that may *not* be disregarded in a mental state μ must satisfy. Such a set of transitions is called an *ample set*, and different ample sets may exist in the same mental state. In Sect. 10.6 of [24], it is proven that model checking results are correct when the reduced transition system, obtained by only executing transitions in ample sets (thus satisfying the four conditions), is verified instead of the full transition system.

Let $\mathsf{Ample} : \Omega_M \rightarrow 2^{\Omega_T}$ be a function that maps a mental state μ to a subset of the transitions that are enabled in μ, i.e. $\mathsf{Ample}(\mu) \subseteq \mathsf{En}(\mu)$, such that the four conditions in Table 7.1 are satisfied, i.e. $\mathsf{Ample}(\mu)$ is an ample set in μ. In the remainder of this and the subsequent section, we discuss ways to implement this function; as multiple ample sets may exist in a mental state, various implementations of Ample are possible, potentially yielding different ample sets. A trivial implementation is one that maps μ to *all* transitions that are enabled in it. In such case, the full transition system is obtained, hence no reduction is yielded. We will try to develop better implementations, and proceed as follows. In the remainder of this subsection, we elaborate on the four ample set conditions, i.e. their meaning and computational complexity. Later, in Sect. 7.2, two implementations of Ample such that the conditions are satisfied are discussed.

Ample set conditions

Table 7.1 lists the four conditions that implementations of Ample must satisfy.

Condition **C0** guarantees that if a mental state has at least one successor in the full transition system, then it also has at least one successor in the reduced transition system.

The implications that satisfaction of **C1** has are more complex: by Lemma 24 in [24], satisfaction of **C1** implies that the transitions in $\mathsf{En}(\mu) \setminus \mathsf{Ample}(\mu)$ are all independent of the transitions in $\mathsf{Ample}(\mu)$. Informally, the proof by contradictions is as follows. Suppose $\tau \in \mathsf{Ample}(\mu)$, $\tau' \in \mathsf{En}(\mu) \setminus \mathsf{Ample}(\mu)$, and τ, τ' are dependent. Because τ' is enabled in μ, there is a path starting

C0 – Emptiness $\mathsf{Ample}(\mu) = \emptyset$ iff $\mathsf{En}(\mu) = \emptyset$.	**C2 – Invisibility** Every $\tau \in \mathsf{Ample}(\mu)$ is not visible to the property under investigation, unless $\mathsf{Ample}(\mu) = \mathsf{En}(\mu)$.
C1 – Ample decomposition In the full transition system, on any path starting from state μ, a transition dependent* on a transition from $\mathsf{Ample}(\mu)$ cannot appear before some transition from $\mathsf{Ample}(\mu)$ is executed.	**C3 – Cycle closing** If a cycle contains a state in which a transition τ is enabled, then it also contains a state μ such that $\tau \in \mathsf{Ample}(\mu)$.

* According to the dependence relation in Def. 27 of Sect. 5.4.3 .

Table 7.1: Conditions that implementations of Ample must satisfy in each mental state in the reduced transition system to ensure that the model checker reports correct results.

in μ in the full transition system that starts with execution of τ'. Then, a transition dependent on a transition in $\mathsf{Ample}(\mu)$ (namely τ') is executed before a transition in $\mathsf{Ample}(\mu)$, violating **C1**. However, we already know that $\mathsf{Ample}(\mu)$ is an ample set, hence satisfies **C1**, which yields a contradiction.

To describe the purpose of condition **C2**, suppose we would only impose conditions **C0** and **C1**. Then, there are two ways for the reduced transition system to *not* contain a path $\pi = \mu \cdots$ starting in μ that *does* exist in the full transition system, and that is essential for correctness to be preserved:

- *The path π has a prefix $\pi_{pre} = \mu\mu' \cdots \mu^{(n)}$ such that there exists a $\tau_{n-1} \in \mathsf{Ample}(\mu)$ such that $\langle \mu^{(n-1)}, \mu^{(n)} \rangle \in \tau_{n-1}$ and for all $0 \leq i < n-1$, there exists a τ_i such that $\langle \mu^{(i)}, \mu^{(i+1)} \rangle \in \tau_i$ and that is independent of all transitions in $\mathsf{Ample}(\mu)$ by Lemma 24 of [24].*

  In other words, π_{pre} is generated by execution of the transitions $\tau_0 \cdots \tau_{n-2}\tau_{n-1}$ such that $\tau_{n-1} \in \mathsf{Ample}(\mu)$ and each τ_i is independent of all transitions in $\mathsf{Ample}(\mu)$ for all $0 \leq i < n-1$. Although a path with this prefix does not exist in the reduced transition system, it does contain a path π' with a prefix $\pi'_{pre} = \mu \cdots \mu^{(n)}$ that is generated by execution of the transition sequence $\tau_{n-1}\tau_0 \cdots \tau_{n-2}$, and that ends in the same state $\mu^{(n)}$ as π_{pre}. To see this, note that τ_{n-1} is independent with all the transitions that were executed before it. Consequently, due to the enabledness and commutativity conditions (see Sect. 5.4.3), it can be shifted all the way to the beginning of the sequence. Figure 7.3 shows this graphically. Because the prefixes π_{pre} and π'_{pre} both end in the same mental state $\mu^{(n)}$, π' can be regarded as a representative of π. However, for the removal of π by the existence of π' to be justified, their two prefixes π_{pre} and π'_{pre} should also be indistinguishable with respect to the property under investigation.

  To ensure this, condition **C2** is imposed: it implies that execution of τ_{n-1} (which is in $\mathsf{Ample}(\mu)$) does not change a mental state condition occurring in the property under investigation. Thus, this property will not be able to distinguish between prefixes π_{pre} and π'_{pre} (e.g. [24]).

- *The path $\pi = \mu\mu' \cdots$ is an infinite sequence such that for all $i \geq 0$, there exists a τ_i such that $\langle \mu^{(i)}, \mu^{(i+1)} \rangle \in \tau_i$ and that is independent of all transitions in $\mathsf{Ample}(\mu)$ by Lemma 24 of [24].*

  In other words, π is generated by execution of the transitions $\tau_0\tau_1 \cdots$ such that each τ_i is independent of all transitions in $\mathsf{Ample}(\mu)$ for all $i \geq 0$. Although π does not exist in the reduced transition system (as assumed above), the transition system does contain a path π' that is generated by execution of the transition sequence $\tau\tau_0\tau_1 \cdots$ in which $\tau \in \mathsf{Ample}(\mu)$. To see this, note that each τ_i is independent of all transitions in $\mathsf{Ample}(\mu)$ (including τ) such that execution of τ cannot disable any such τ_i. However, for the removal of π by the existence of π' to be justified, the property under investigation should be invariant to the execution of τ. To ensure this, condition **C2** is imposed.

Finally, condition **C3** guarantees that execution of a transition τ that is not in $\mathsf{Ample}(\mu)$, i.e. $\tau \in \mathsf{En}(\mu) \setminus \mathsf{Ample}(\mu)$, is not *deferred forever on a cycle*. To illustrate this problem, called the *ignoring problem* (first recognised in [96]), consider Fig. 7.4 and suppose **C3** is not imposed. In the full transition system in Fig. 7.4a, τ' is independent of τ_0, τ_1. Suppose τ_0, τ_1 are invisible to the property, e.g. $\mathsf{F}\psi$, while τ' is visible, e.g. changes the truth value of Msc ψ from false to true. We will now construct a reduced transition system, starting from the initial mental state μ_0. In μ_0, we can select $\mathsf{Ample}(\mu_0) = \{\tau_0\}$, because this set satisfies **C0** ($\mathsf{Ample}(\mu_0) \neq \emptyset$), **C1** ($\tau_0$ is independent of τ'), and **C2** (τ_0 is invisible). After executing τ_0, we arrive in μ_1. In this mental state, we can select $\mathsf{Ample}(\mu_1) = \{\tau_1\}$, because this set satisfies **C0**, **C1**, and **C2**. Execution of τ_1 in μ_1 leads us back to μ_0, which already has been visited. Subsequently, the construction terminates and we obtain the reduced transition system shown in Fig. 7.4b. However, the reduced transition system does not contain a path on which ψ eventually becomes true. That is, during

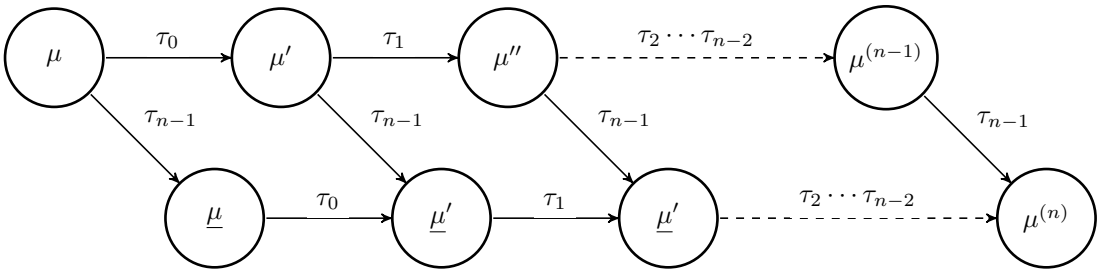
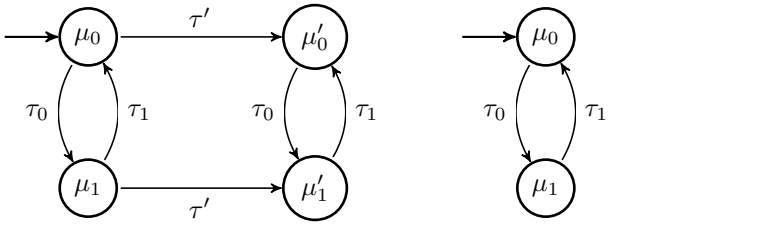


Figure 7.3: Example showing how transition τ_{n-1} can be shifted to the beginning of the sequence if it is independent with all transitions on the sequence $\tau_0 \cdots \tau_{n-2}$.



(a) Full transition system (b) Reduced transition system

Figure 7.4: Ignoring problem.

the construction, each mental state on the cycle $\mu_0\mu_1\mu_0$ has deferred execution of τ' to a possible future mental state, but when the cycle is closed, the construction terminates and τ' is ignored. Condition **C3** prevents this from happening.

As aforementioned, it has been proven (e.g. in [24]) that implementations of Ample that adhere to these four conditions constitute correct persistent sets, i.e. when Ample is used instead of En, sufficiently many computations are preserved for the model checker to produce a correct result. We justify our choice for the ample set method in Sect. 7.1.2. That section treats other approaches to partial order reduction as well. Before that, we discuss some computational properties of checking the ample set method's four conditions.

Computational complexity

Recall that in the introduction of Sect. 7.1, we listed three properties that the use of persistent sets should satisfy: the first property concerned correctness, while the second and third property concerned the trade-off between computational complexity and effectiveness of the reduction. With the ample set method, the first property is satisfied by the specification of the four conditions. We address the second and third property by discussing the (theoretical) computational complexity of checking these conditions.

Checking **C0** is easy: we can do that in constant time [24].

Condition **C1** is a lot more difficult to check. The reason is that we need information not only about the current mental state μ, but about all the states in the *full transition system* that are reachable from μ. In other words, checking **C1** requires us to first generate the full transition system. In fact, it has been proven (e.g. in [24]) that checking **C1** for a mental state μ and a subset of $\mathsf{En}(\mu)$ is at least as hard as the model checking problem for the full transition system. Obviously, generating the full transition system for the sake of generating a reduced transition system is not an option in practice. Thus, rather than checking **C1** for an arbitrary subset of enabled transitions, we will apply a heuristic approach that finds a set of transitions that is guaranteed to satisfy **C1**, similar to existing implementations of the ample set method (e.g. [58]). Such an approach does

not always lead to an ample set that achieves the greatest reduction possible, but can be quite effective nevertheless [24].

Checking **C2** is, in contrast to **C1**, relatively easy as it concerns a local constraint that can be checked by analysing the transitions that are enabled in the current mental state. Recall that in Sect. 5.4.1, we already introduced a method for establishing visibility of a transition τ with respect to an LTL formula ϕ, as well as for transition classes. Although the latter method over-approximates visibility in the sense that in some cases, a transition belonging to a transition class is deemed visible when it actually is not, this does not violate correctness of the ample set method. Because if this were to happen, **C2** is not satisfied by the *candidate ample set* of which τ is a member, and consequently, this candidate set is discarded. Then, in the worst case, no other ample set is found such that all enabled transitions need be executed, but correctness of the algorithm is never at stake.

Finally, checking **C3** is more difficult again. Like **C1**, **C3** is defined in global terms, but unlike **C1**, **C3** refers to the reduced transition system rather than to the full transitions system. It has been proven (e.g. in [24]) that a sufficient condition for **C3** is that at least one mental state along each cycle is fully expanded, i.e. all successors of this mental state are explored (or will be). Then, we can strengthen **C3** to the condition **C3′** [24]:

C3′ – Cycle closing for depth-first search
If μ is not fully expanded, then no transition in $\mathsf{Ample}(\mu)$ may reach a mental state that is on the search stack.

C3′ can straightforwardly be checked by matching the successors that a set of enabled transitions generates to the mental states on the search stack. In the worst case, this check takes time linear in the size of the reduced state space, but when hashing data structures are used, this can be done much faster, i.e. in constant time if mental states are divided uniformly over buckets.

7.1.2 Related Work

We favour the ample set method for the following reasons. First, because the ample set method is defined as a set of conditions rather than as a specific algorithm tailored to verification of concurrent systems, it can be applied to GOAL in a clean fashion. That is, even though the ample set method was conceived for the verification of concurrent systems, its four conditions are generic in the sense that the notion of concurrency does not occur in any of them. A second reason is that the ample set method has extensively been described in the literature by different authors. This contributed to the understanding of the theory, and the ability to implement it for GOAL. Also, the method has successfully been applied in the SPIN model checker, whereas the status of other approaches is not always clear. Fourth, the ample set method is, with some additional constraints [60], compatible with on-the-fly model checking.

Below we discuss some work in the field of partial order reduction that is directly related to (other persistent set methods) or compatible with (sleep sets) the ample set method. For completeness, we also list other POR techniques that are not compatible with persistent sets; a comprehensive overview of partial order reduction is beyond the scope of this thesis. Finally, we discuss recent research on POR methods for agent verification.

Other persistent sets

The ample set method is not the only method for computing persistent sets. In [96], an alternative algorithm is proposed, in which the persistent set to be computed is called a *stubborn set*. The algorithm for computation of these stubborn sets involves the construction of a graph, whose vertices are transitions and whose edges are defined by the dependence relation on transitions. To find the stubborn set, the algorithm searches this graph for strongly connected components that include the set of visible transitions. In [46], an improvement to the stubborn set method is proposed. This method tries to produce smaller stubborn sets, called *conditional stubborn sets*,

by refining the dependence relation between transitions. Hence, it is not a fundamentally different algorithm.

Sleep sets

The sleep set method (e.g. [46]) uses a quite different approach to find a subset of enabled transitions: it tries to take advantage of information about the past of the search. For each state s that has already been visited during the search, a set of transitions $sleep(s)$ is maintained, which contains transitions that do not need to be traversed from s. The intuition behind such a set is as follows. Suppose that a depth-first search algorithm has just backtracked from a successor state $\alpha(s)$ of s that was reached after execution of a transition α. Let β be another enabled transition in s that has not yet been executed, and let α and β be independent. After backtracking from $\alpha(s)$, the algorithm can traverse β, and continue exploration from $\beta(s)$. Because α and β are independent with each other, α is also enabled in $\beta(s)$. However, execution of α in $\beta(s)$ is unnecessary, as it leads to a state that has already been sufficiently investigated (otherwise the algorithm could not have backtracked from $\alpha(s)$ yet). That is, α is said to be in $\beta(s)$'s sleep set, and as such will not be re-executed. Note that the sleep set method reduces the number of transitions rather than the number of states. Also, the sleep set method and persistent set methods are complementary such that an implementation of the sleep set method for Goal could be among future extensions to the Goal model checker.

Other approaches

In [74, 38], a partial order reduction technique based on *net unfoldings* is presented, in which concurrent systems are modelled as Petri nets. More recently, in [42], *dynamic partial order reduction* is introduced, in which interaction between different processes is tracked dynamically. The method can, however, only be used for detection of deadlocks and violation of safety properties, and is only applicable to acyclic state spaces. Finally, *cartesian partial order reduction*, introduced in [48], is a method that tries to identify sequences of transitions within the same process that can be executed atomically using *cartesian vectors*.

Partial order reduction in agent verification

To the best of our knowledge, research on partial order reduction in agent verification is limited to the publication of two papers from the same research group: [73] and [72]. The former paper presents theoretical results; the latter paper extends this work with an actual algorithm and experimental results obtained from its implementation. Although both the [72] algorithm and our approach treat Por in the context of agent verification, there are differences. First, the [72] algorithm is designed for the verification of (concurrent) multi-agent systems, whereas we consider single-agent systems; [72] applies Por in the traditional sense, i.e. for the sake of reducing the number of different orders of concurrent events. Second, the verification logic in [72] contains an epistemic operator only (whose semantics is different from Goal's belief operator), and no operator to express motivational attitudes; moreover, this epistemic operator is a first-class citizen of the logic, whereas in our verification logic, we treat mental state conditions as propositions. Third, the [72] algorithm seems to aim at the verification of *models* of agent systems rather than on the verification of actual agent *programs*, which is the target of our algorithm. On the one hand, this might make the [72] algorithm applicable to a larger range of agent systems, while one the other, one could argue that many actual agent programs cannot straightforwardly be specified in the specification formalism that is assumed in [72]. There are similarities as well. Both the algorithm of [72] and our approach are based on the ample set method, and both are applied under a depth-first search regime.

7.2 Implementation

In this section, we treat the implementation of the ample set method for the verification of GOAL agents. We first discuss heuristics for finding a subset of enabled transitions that are guaranteed to satisfy **C1**. Next, we describe an algorithm, derived from Sect. 10.5.2 of [24], that given these heuristics computes ample sets. Finally, we treat the application of the algorithm in an on-the-fly approach.

7.2.1 Heuristics for C1

As argued in Sect. 7.1.1, it is not practical to determine for a subset of enabled transitions in a mental state μ whether it satisfies **C1**, as performing this check is at least as hard as the model checking problem itself (see also Theorem 11 in [24]). Thus, rather than checking **C1** for such a subset of $\mathsf{En}(\mu)$, we heuristically compute a subset of $\mathsf{En}(\mu)$ that is *guaranteed* to satisfy **C1**. We call such a subset a *candidate ample set*, and denote the set of all such candidates in μ by $\mathsf{Cands}(\mu)$. The two heuristics that we present in the following, denoted $\mathrm{H}^{sing}_{\mathrm{POR}}$ and $\mathrm{H}^{mult}_{\mathrm{POR}}$ (the meaning of the superscripts is treated shortly), describe two different ways to compute $\mathsf{Cands}(\mu)$ such that the candidate ample sets it contains satisfy **C1**. The heuristics are tailored to GOAL, and, similar to the heuristics applied in SPIN's POR algorithm [58], conservative: the candidate ample sets that the heuristics generate are relatively cheap to compute during verification, at the cost of overlooking or discarding subsets of $\mathsf{En}(\mu)$ that are actually ample (recall from Sect. 7.1.1 that different ample sets may exist in μ).

The reason for this conservative approach is that we want to minimise the overhead of ample set computation as much as possible. To this end, we use the approximate enabled-by relation $\mathsf{ApproxEnabledBy}$ (defined in Def. 25) and the approximate dependence relation $\mathsf{ApproxDep}$ (defined in Def. 29) such that most of the computations that the POR algorithm requires can be done before the generation of the transition system commences, i.e. *off-line* (this is also done by SPIN [24]). Recall that both these relations are over-approximations of the precise relations $\mathsf{EnabledBy}$ and Dep; this is one of the reasons why ample sets are sometimes discarded as mentioned above. The fact that use of such over-approximations does not violate correctness is proven in two propositions (one for each heuristic) presented below.

Because we want to do most of the computations required for the POR algorithm off-line using the $\mathsf{ApproxDep}$ and $\mathsf{ApproxEnabledBy}$ relations, we generalise over transitions by means of transition classes. That is, we compute these approximate relations before generation of the transition system, and assume the implied Dep and $\mathsf{EnabledBy}$ relations for the transitions belonging to the transition classes (instead of computing the relations more precisely for individual transitions) during generation of the transition system. Because all transitions belonging to the same transition class are treated equally during generation, we define the heuristics in terms of transition classes.

In the remainder of this subsection, we first introduce $\mathrm{H}^{sing}_{\mathrm{POR}}$ and $\mathrm{H}^{mult}_{\mathrm{POR}}$. Then, at the end of this subsection, we compare them with the heuristic used in SPIN.

Heuristic $\mathrm{H}^{sing}_{\mathrm{POR}}$

The first heuristic that we discuss, i.e. $\mathrm{H}^{sing}_{\mathrm{POR}}$, yields candidate ample sets by considering *sing*le transition classes in isolation. That is, it yields candidate sets that contain transitions that all belong to the same transition class. This requires iteration over all transition classes to determine whether they constitute a candidate set in each mental state. We discuss how this works for a mental state μ and transition class $\boldsymbol{\tau}(\rho)$. Basically, $\mathrm{H}^{sing}_{\mathrm{POR}}$ puts all enabled transitions in μ that belong to $\boldsymbol{\tau}(\rho)$ in a candidate set T, i.e. $T = \boldsymbol{\tau}(\rho) \cap \mathsf{En}(\mu)$. However, not all transition classes that we would treat as such yield satisfactory candidate ample sets: **C1** may be violated in two cases. In both cases some transitions independent of those in T are executed, eventually enabling a transition τ that is dependent on those in T.

1. In the first case, τ belongs to a transition class $\boldsymbol{\tau}(\rho')$ such that $\boldsymbol{\tau}(\rho') \neq \boldsymbol{\tau}(\rho)$. To prevent this from happening, we should check that all transitions in $\boldsymbol{\tau}(\rho)$ are independent from all

transitions in other transition classes. If this is not the case, the set of enabled transitions in $\boldsymbol{\tau}(\rho)$ is discarded as a candidate ample set.

2. In the second case, τ belongs to $\boldsymbol{\tau}(\rho)$ itself. Then, the transitions independent from those in T that were executed prior to τ have caused τ to become enabled (if τ already was enabled before execution of any of those transitions, τ would be in T). To prevent this from happening, we should check that all transitions in $\boldsymbol{\tau}(\rho)$ cannot be enabled by transitions that do not belong to $\boldsymbol{\tau}(\rho)$. If this is not the case, the set of enabled transitions in $\boldsymbol{\tau}(\rho)$ is discarded as a candidate ample set.

The above two cases lead to the formulation of two conditions that a transition class $\boldsymbol{\tau}(\rho)$ must satisfy for its enabled transitions in μ to qualify as a candidate ample set: they must be independent from all transitions belonging to other transition classes, and the execution of transitions belonging to other transition classes may not enable them. More formally, let $\boldsymbol{\Omega_T}$ be the set of all transition classes, let $\boldsymbol{\tau}(\rho) \in \boldsymbol{\Omega_T}$, and let μ be a mental state. Then, $\boldsymbol{\tau}(\rho) \cap \mathsf{En}(\mu)$ is a set of transitions that satisfies **C1** if:

- for all $\boldsymbol{\tau}(\rho') \in \boldsymbol{\Omega_T} \setminus \{\boldsymbol{\tau}(\rho)\}$, neither $\mathsf{ApproxDep}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$ nor $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$

We already remark at this point that there are very strong conceptual similarities between $\mathrm{H}^{sing}_{\mathrm{POR}}$ and the heuristic used in Spin; we elaborate on these at the end of this subsection.

Proposition 8 in Appx. C.4 establishes that with the above conditions, the candidate ample sets that $\mathrm{H}^{sing}_{\mathrm{POR}}$ yield indeed satisfy **C1**. Computation of these candidate ample sets in practice is straightforward; the algorithm for doing so occurs as Alg. 4. Note that $\mathsf{ApproxDep}$ and $\mathsf{ApproxEnabledBy}$ can be computed off-line, reducing much of the work during generation.

Heuristic $\mathrm{H}^{mult}_{\mathrm{POR}}$

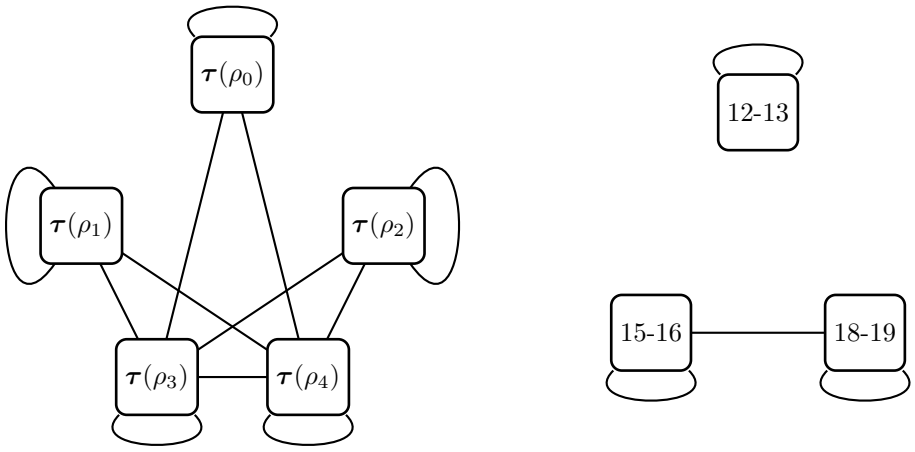
The second heuristic, denoted $\mathrm{H}^{mult}_{\mathrm{POR}}$, is a generalisation of $\mathrm{H}^{sing}_{\mathrm{POR}}$: while the candidate ample sets computed with $\mathrm{H}^{sing}_{\mathrm{POR}}$ are always subsets of a single transition class, $\mathrm{H}^{mult}_{\mathrm{POR}}$ combines transitions from *mult*iple transition classes to construct candidate sets.

Informally, $\mathrm{H}^{mult}_{\mathrm{POR}}$ picks a set of transition classes $\boldsymbol{T}$, and puts all their enabled transitions in a mental state μ in the candidate ample set T, i.e. $T = \mathsf{En}(\mu) \cap \bigcup_{\boldsymbol{\tau}(\rho) \in \boldsymbol{T}} \boldsymbol{\tau}(\rho)$. However, similar to $\mathrm{H}^{sing}_{\mathrm{POR}}$, not any set of transition classes $\boldsymbol{T}$ that we would treat as such yields satisfactory candidate ample sets: in fact, **C1** may be violated in the same two cases as outlined when treating $\mathrm{H}^{sing}_{\mathrm{POR}}$. Therefore, similar conditions are imposed on a set $\boldsymbol{T}$. Formally, we have the following. Let $\boldsymbol{\Omega_T}$ be the set of all transition classes, let $\boldsymbol{T} \subseteq \boldsymbol{\Omega_T}$, let $T = \bigcup_{\boldsymbol{\tau}(\rho) \in \boldsymbol{T}} \boldsymbol{\tau}(\rho)$ be a set of transitions, and let μ be a mental state. Then, $T \cap \mathsf{En}(\mu)$ is a set of transitions that satisfies **C1** if:

- for all $\boldsymbol{\tau}(\rho') \in \boldsymbol{\Omega_T} \setminus \boldsymbol{T}$, there does not exist a $\boldsymbol{\tau}(\rho) \in \boldsymbol{T}$ such that $\mathsf{ApproxDep}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$ or $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$

1: $\mathsf{Cands} := \emptyset$
2: **for all** $\boldsymbol{\tau}(\rho) \in \boldsymbol{\Omega_T}$ **do**
3: **for all** $\boldsymbol{\tau}(\rho') \in \boldsymbol{\Omega_T} \setminus \{\boldsymbol{\tau}(\rho)\}$ **do**
4: **if** $\mathsf{ApproxDep}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$ **or** $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$ **then**
5: break the inner **for** loops, and continue the outer **for** loop with the next $\boldsymbol{\tau}(\rho)$
6: **end if**
7: **end for**
8: $\mathsf{Cands} := \mathsf{Cands} \cup \{\boldsymbol{\tau}(\rho) \cap \mathsf{En}(\mu)\}$
9: **end for**

Algorithm 4: Let $\boldsymbol{\Omega_T}$ be the set of all transition classes, and let μ be a mental state. Then: this algorithm computes a set of candidate ample sets Cands according to $\mathrm{H}^{sing}_{\mathrm{POR}}$.



(a) `blenderAgent` (b) `blocksCounterAgent`

Figure 7.5: Dependence graphs for transition classes of `blenderAgent` and `blocksCounterAgent`.

If $\boldsymbol{T}$ is a singleton, i.e. contains only a single transition class, then $\mathrm{H}^{mult}_{\mathrm{POR}}$ is reduced to $\mathrm{H}^{sing}_{\mathrm{POR}}$. Proposition 10 in Appx. C.4 establishes that candidate ample sets yielded by $\mathrm{H}^{mult}_{\mathrm{POR}}$ satisfy **C1**.

The challenge with $\mathrm{H}^{mult}_{\mathrm{POR}}$ in practice is finding suitable sets $\boldsymbol{T}$. A straightforward approach is iterating over all elements in the power set of $\boldsymbol{\Omega_T}$ and check the above condition for each such $\boldsymbol{T} \in 2^{\boldsymbol{\Omega_T}}$, but as this requires time exponential in the number of transition classes, it is unfavoured. Instead, we let the search for sets $\boldsymbol{T}$ be guided by the definition of ApproxDep. That is, we search for sets $\boldsymbol{T}$ that are guaranteed to satisfy the requirement that transition classes in $\boldsymbol{T}$ are independent of all transition classes outside $\boldsymbol{T}$. This search can be done in time linear in the number of transition classes n and the size of ApproxDep, and yields at most n sets $\boldsymbol{T}$ instead of 2^n. It works as follows.

To find sets $\boldsymbol{T}$ that satisfy the first requirement, we regard the dependence relation on transition classes ApproxDep as an undirected graph whose vertices are transition classes, and whose edges are elements of the relation. Examples of such graphs for `blenderAgent` and `blocksCounterAgent` are given in Fig. 7.5. The labels on vertices in `blenderAgent`'s graph correspond to transition classes as listed in Fig. 6.2; the labels on vertices in `blocksCounterAgent`'s graph correspond to transition classes whose action rule is defined on the referenced line numbers in Fig. 6.3. Next, we observe that because every edge corresponds to an element of ApproxDep, every transition class belonging to a set $\boldsymbol{T}$ may only have edges to transition classes belonging to $\boldsymbol{T}$, but not to others. That is, a set $\boldsymbol{T}$ that satisfies the first requirement of the condition above corresponds to a *connected component* of the graph. For example, in Fig. 7.5a, there is only a single connected component, namely the entire graph. In Fig. 7.5b, there are two, namely $\{12\text{-}13\}$ and $\{15\text{-}16, 18\text{-}19\}$. Connected components can be found with a depth-first search (e.g. [61]), which runs in time linear in the number of vertices and edges. Because there cannot be more connected components than vertices in the graph, indeed this approach yields at most n sets $\boldsymbol{T}$ (where n is the number of transition classes) as aforementioned.

An algorithm for the computation of candidate ample sets using $\mathrm{H}^{mult}_{\mathrm{POR}}$ is given as Alg. 5; the algorithm is quite similar to Alg. 4. We remark that computation of connected components can occur before actual verification commences, and is as such a one-time-expense. Consequently, the overhead during the nested depth-first search is still small as desired.

Comparison with Spin

In the Spin implementation (e.g. [59, 24]), the heuristic used to compute sets satisfying **C1** is in essence quite similar to our $\mathrm{H}^{sing}_{\mathrm{POR}}$. Recall that Spin is a model checker for the verification of

concurrent imperative systems that consist of multiple processes. Whereas our $\mathrm{H}^{sing}_{\mathrm{POR}}$ yields ample sets that are constituted by transitions belonging to the same transition class (i.e. generated by the same action rule), the heuristic used in Spin yields ample sets that are constituted by transitions belonging to the same process. The transitions belonging to a candidate ample set in Spin (which thus all belong to the same process) must satisfy similar conditions as prescribed by $\mathrm{H}^{sing}_{\mathrm{POR}}$: they must be independent with all transitions belonging to other processes, and there may not exist transitions belonging to other processes that can enable them. Furthermore, in Spin, the transitions belonging to the same process are assumed *interdependent* (i.e. each transition depends on all other transitions belonging to the same process), which is also the case for the transitions belonging to the same transition class when computed with the approximate dependence relation as outlined at the end of Sect. 5.4.3. Thus, we may regard $\mathrm{H}^{sing}_{\mathrm{POR}}$ as Spin's heuristic in which each action rule corresponds to a separate process. An equivalent to heuristic $\mathrm{H}^{mult}_{\mathrm{POR}}$, in contrast, does not occur in the Spin literature nor have we found it in any other publication on the ample set method.

7.2.2 Algorithm

Given heuristics $\mathrm{H}^{sing}_{\mathrm{POR}}$ and $\mathrm{H}^{mult}_{\mathrm{POR}}$, which only provide us with candidate ample sets whose satisfaction of **C0**, **C2**, and **C3** is not checked yet, the ample set algorithm appears as Alg. 6.

It works as follows. On line 1, the ample set to be computed is initialised as empty, and on line 2, the candidate sets are computed based on $\mathrm{H}^{sing}_{\mathrm{POR}}$ or $\mathrm{H}^{mult}_{\mathrm{POR}}$. The loop on lines 3-12 iterates over all these candidate sets. If no ample set has been found yet (line 4), the transition classes belonging to the current candidate set are iterated (lines 5-9) and checked for violation of **C2** and **C3′** on line 6. That is, the first disjunct is true if the current transition class violates **C2** (we may check $\mathsf{ApproxVisible}(\boldsymbol{\tau}(\rho), \varphi)$ instead of $\mathsf{Visible}(\tau, \varphi)$ by Theorem 4), while the second disjunct is true if the current disjunct violates **C3′**. In such cases, the inner **for** loops are broken off, because $\boldsymbol{T}$ can never yield a proper ample set in such case. In contrast, if line 10 is ever reached, then no violation of **C2** or **C3′** has occurred, hence an ample set is found. Finally, if a non-empty ample set cannot be found, the algorithm sets $\mathsf{Ample}(\mu)$ to the set of all enabled transitions in mental state μ (line 13-15). This ensures satisfaction of **C0**.

As aforementioned, the algorithm is derived from the algorithm given in Sect. 10.5.2 of [24]; the main contribution here is the application to Goal, specifically the heuristics used to ensure satisfaction of **C1** combined with computation of the enabled-by and dependence relations (as

1: $\mathsf{Cands} := \emptyset$
2: $\mathsf{ConnComps}$:= the set of connected components in the dependence graph for transition classes, computed by depth-first search (e.g. [61])
3: **for all** $\boldsymbol{T} \in \mathsf{ConnComps}$ **do**
4: **for all** $\boldsymbol{\tau}(\rho) \in \boldsymbol{T}$ **do**
5: **for all** $\boldsymbol{\tau}(\rho') \in \boldsymbol{\Omega_T} \setminus \boldsymbol{T}$ **do**
6: **if** $\mathsf{ApproxDep}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$* **or** $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$ **then**
7: break the inner two **for** loops, and continue the outer **for** loop with the next $\boldsymbol{T}$
8: **end if**
9: **end for**
10: **end for**
11: $T := \bigcup_{\boldsymbol{\tau}(\rho) \in \boldsymbol{T}} \boldsymbol{\tau}(\rho)$
12: $\mathsf{Cands} := \mathsf{Cands} \cup \{ T \cap \mathsf{En}(\mu) \}$
13: **end for**

* This check is actually superfluous due to the way connected components are computed, but stated here nevertheless to let the algorithm correspond more straightforwardly to the condition formulated for $\mathrm{H}^{mult}_{\mathrm{POR}}$.

Algorithm 5: Let $\boldsymbol{\Omega_T}$ be the set of all transition classes, and let μ be a mental state. Then: this algorithm computes a set of candidate ample sets Cands according to $\mathrm{H}^{mult}_{\mathrm{POR}}$.

treated in Chap. 5).

7.2.3 On-The-Fly Partial Order Reduction

The ample set method is compatible with on-the-fly exploration of the state space. However, it does require additional bookkeeping to ensure that the *same* ample set is generated if a state is revisited during the nested depth-first search [24]. That is, if a different ample set would be chosen each time the same state is visited, the correctness of the model checking algorithm may be violated (see [60] for a counterexample).

In the literature, various methods to implement this requirement have been proposed (e.g. [60]). The most simple approach is to fix the order in which ample sets are computed, and always return the first one that is found. A second option is to fix the order, and store with each state an index i, indicating that the i-th computed ample set was chosen the first time the state was encountered.

We note that to minimise the overhead of our POR computation, Alg. 6 does not compute multiple ample sets but only one (i.e. as soon as one non-empty ample set is found, no other ample sets are computed by means of the condition on line 3). Consequently, the second option is not relevant for us. As a side note, however, if we would compute multiple ample sets, it is not directly obvious how to pick one: although it is tempting to always choose the smallest ample set, it can be shown that the smallest such set does not necessarily yield the greatest reduction possible [96]. Thus, we choose the first option, and fix the order in which ample sets are computed.

7.3 Case Studies

In this section, we investigate whether the POR algorithm presented in this chapter is able to effectively reduce the state space of a transition system. First, we revisit the `blenderAgent` example with which we started and motivated this chapter. Subsequently, we treat an example involving `blocksCounterAgent` (introduced in Sect. 6.4.2 of the previous chapter). The section is concluded with a discussion.

1: $\mathsf{Ample}(\mu) := \emptyset$
2: $\mathsf{Cands}(\mu) :=$ a set of candidate ample sets according to $\mathrm{H}^{sing}_{\mathrm{POR}}$ (Alg. 4) or $\mathrm{H}^{mult}_{\mathrm{POR}}$ (Alg. 5)

3: **for all** $T \in \mathsf{Cands}(\mu)$ **do**
4: **if** $\mathsf{Ample}(\mu) = \emptyset$ **then**
5: **for all** $\tau \in T$ such that there exists $\boldsymbol{\tau}(\rho) \in \boldsymbol{\Omega_T}$ such that $\tau \in \boldsymbol{\tau}(\rho)$ **do**
6: **if** $\mathsf{ApproxVisible}(\boldsymbol{\tau}(\rho), \varphi)$ **or** $\tau(\mu) \in M_\pi$ **then**
7: break the inner **for** loop, and continue the outer **for** loop with the next T
8: **end if**
9: **end for**
10: $\mathsf{Ample}(\mu) := T$
11: **end if**
12: **end for**

13: **if** $\mathsf{Ample}(\mu) = \emptyset$ **then**
14: $\mathsf{Ample}(\mu) := \mathsf{En}(\mu)$
15: **end if**

Algorithm 6: Let μ be a mental state, let M_π be the current search path, and let φ be the property under investigation. Then: this algorithm computes a single ample set $\mathsf{Ample}(\mu)$.

7.3.1 `blenderAgent`

Let us revisit the `blenderAgent` example. In the chapter's introduction, we argued that for verifying whether `blenderAgent` eventually believes to have filled the blender, i.e. $\varphi = \mathsf{F}\,\texttt{bel(filled)}$, it is sufficient to investigate only one representative computation instead of all 16 (see also Fig. 7.1). Here, we investigate whether the algorithm and heuristics for **C1** introduced in the previous section allow for such a reduction. We will first see that for the example in the introduction, no reduction is gained, and argue why this is the case. Subsequently, we discuss another example that concerns a buggy version of `blenderAgent`. In that example, the Por algorithm is able to reduce the transition system, because the bugs cause less dependences to be present. As such, more transition classes are independent, which has a positive effect on reduction results.

Correct `blenderAgent`: no reduction

Unfortunately, the Por algorithm does not reduce the transition system with either $\mathrm{H}^{sing}_{\text{Por}}$ or $\mathrm{H}^{mult}_{\text{Por}}$ when `blenderAgent` is under verification with respect to φ as in the introduction. In fact, the verification run takes longer to finish with Por than without Por due to the overhead that the algorithm imposes. How can this be? Well, although `blenderAgent`'s subtasks of putting bananas and oranges into the blender are independent of each other, the subtasks that precede (i.e. washing the fruit) and succeed these (i.e. ticking check boxes off the recipe and switching on the blender) are not. Although we *know* that `blenderAgent` cannot put fruit into the blender before it has washed the fruit, and although we *know* that `blenderAgent` cannot tick check boxes and switch on the blender before it has put sufficiently many fruit into the blender, the static source code analysis that the model checker carries out to compute the (in)dependence relations is inadequate in this respect. For example, this analysis concludes that switching on the blender is dependent on putting bananas into the reservoir: switching on the blender may achieve `blenderAgent`'s goal such that all transitions that cause fruit to be added become disabled. Although we *know* that by the time `blenderAgent` switches on the blender all these transitions are already disabled (because the reservoir is sufficiently filled) such that they cannot *become* disabled by switching on the blender, this cannot be determined by static code analysis.

Buggy `blenderAgent`: reduction

Although in the above, we have stated multiple times that we "know" certain things about the way `blenderAgent` behaves, our knowledge may be incorrect: we *think* we know that `blenderAgent` will not tick a check box before sufficiently many fruit has been put into the blender, but the presence of bugs may interfere with this. Specifically, we now look at undesired behaviour that arises from too much *underspecification* (either deliberately or by accident), a phenomenon in the remainder referred to as *over-underspecification*. We proceed with an example in which Por does prove itself useful when applied to an over-underspecified version of `blenderAgent`. The agent, called `buggyBlenderAgent`, is given in Fig. 7.6 and contains several sources of over-underspecification. We invite the reader to find these by hand without comparing the source code directly with Fig. 2.1.[1]

Before discussing the impact of these errors, let us first reconsider property φ as defined above, whose satisfaction is not affected (correctness of programs is usually not specified as a single comprehensive property, but as several properties addressing different types of behaviour). We have verified this property using both $\mathrm{H}^{sing}_{\text{Por}}$ and $\mathrm{H}^{mult}_{\text{Por}}$ for various values of i in `recipe(bananas,`i`)` and `recipe(oranges,`i`)` in `buggyBlenderAgent`'s knowledge base. The verification statistics are given in Table 7.2 (the "initialisation" row displays the time required to initialise the Por algorithm, i.e compute the necessary relations on transition classes), and show very substantial reductions. In the most extreme case, i.e. for $i = 64$, verification without Por took over three minutes, while

[1] `buggyBlenderAgent` contains the following sources of over-underspecification: on lines 17-19, a **`goal(filled)`** conjunct should be added, the `washed` conjunct is missing on line 25, and in the precondition of the `tick(F)` action on line 33, the conjunct `added(F,Qr)` should be added.

```
main: buggyBlenderAgent{
  knowledge{
    recipe(bananas,2).
    recipe(oranges,2).
    toAdd(F) :- recipe(F,Qr), added(F,Q), Q<Qr.
    filled :- ticked(bananas), ticked(oranges).
  }
  beliefs{
    added(bananas,0).
    added(oranges,0).
    switch(off).
  }
  goals{
    filled, switch(on).
  }
  program{
    if bel(not(washed)) then wash.
    if bel(toAdd(bananas)) then add(bananas,1).
    if bel(toAdd(oranges)) then add(oranges,1).
    if goal(filled), bel(recipe(F,_)) then tick(F).
    if goal(switch(on)), bel(filled) then blend.
  }
  actionspec{
    add(F,Qinc) {
      pre{ added(F,Q), Qnew is Q+Qinc }
      post{ not(added(F,Q)), added(F,Qnew) }
    }
    blend{
      pre{ switch(off) }
      post{ not(switch(off)), switch(on) }
    }
    tick(F){
      pre{ not(ticked(F)), washed, recipe(F,Qr) }
      post{ ticked(F) }
    }
    wash{
      pre{ not(washed) }
      post{ washed }
    }
  }
}
```

Figure 7.6: Buggy version of `blenderAgent`, called `buggyBlenderAgent`.

verification with the POR algorithm enabled finished in less than three seconds after exploring only a very small fraction (less than one percent) of the full transition system. Although readings on memory consumption are indicative as before, a definite (positive) trend is observable: all reductions are within the 25%-50% range. One might notice that there are no significant differences between the use of $\mathrm{H}^{sing}_{\mathrm{POR}}$ and $\mathrm{H}^{mult}_{\mathrm{POR}}$. We come back to this in the next subsection.

How is it possible that the POR algorithm is unable to reduce the transition system of the correct `blenderAgent`, while it yields substantial reductions when we over-underspecify? Well, due to over-underspecifying, `buggyBlenderAgent` is given the opportunity to perform the `tick(F)` action at times when it actually is not allowed to do so, while at the same time, the removal of **`goal(filled)`** conjuncts decouples this first three action rules from performance of `tick(F)`. Consequently, less constraints on when transitions can be executed are present, hence less dependences. The fewer dependences are present, the better POR algorithms in general perform [23] (equivalently, in [96], it is said that systems should be *loosely coupled* for POR to be effective). To see that indeed less dependences are present, Fig. 7.7 shows the dependence graph for transition classes of `buggyBlenderAgent`. Compared to Fig. 7.5a, it is well observable that less dependences are present.

Before turning attention to a property that `buggyBlenderAgent` violates, we like to make two remarks about the previous. First, although one may believe that underspecification of agents is undesirable, this is not the case. On the contrary, underspecification of agent behaviour can have benefits from a design point of view [54]. As such, it is not unlikely that undesired behaviour arising from too much underspecification, like we saw in `buggyBlenderAgent`, occurs in practice (such undesired behaviour should be resolved by refinement of the agent). The second remark that

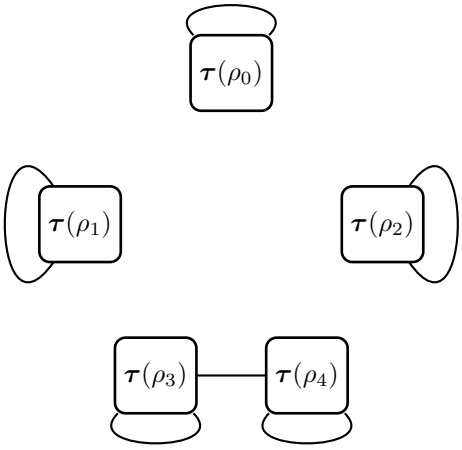


Figure 7.7: Dependence graphs for transition classes of `buggyBlenderAgent`.

we make is that even though `buggyBlenderAgent` is incorrect, it is important that the properties that it *does* satisfy are verified as efficiently as possible: the sooner such properties are reported satisfied, the sooner a property that it violates comes under investigation.

Let us now consider a property that `buggyBlenderAgent` violates. The problem with the missing `added(F,Qr)` conjunct in `tick(F)`'s precondition is that `buggyBlenderAgent` can perform the `tick(F)` action before sufficiently many pieces of `F` have been added. The following properties should uncover this undesired behaviour.

$$\varphi' = \mathsf{G}[\mathbf{bel}(\texttt{ticked(bananas)}) \rightarrow \mathbf{bel}(\texttt{recipe(bananas,Qr),added(bananas,Qr)})]$$
$$\varphi'' = \mathsf{G}[\mathbf{bel}(\texttt{ticked(oranges)}) \rightarrow \mathbf{bel}(\texttt{recipe(oranges,Qr),added(oranges,Qr)})]$$

	Heuristic $\mathrm{H}^{sing}_{\mathrm{POR}}$ for **C1**			Heuristic $\mathrm{H}^{mult}_{\mathrm{POR}}$ for **C1**		
	$i = 4$	$i = 16$	$i = 64$	$i = 4$	$i = 16$	$i = 64$
States						
POR enabled	12	36	132	12	36	132
POR disabled	100	1156	16900	100	1156	16900
Reduction	*88%*	*97%*	*>99%*	*88%*	*97%*	*>99%*
Verification time						
POR enabled	1077 ms	1414 ms	2694 ms	1045 ms	1541 ms	2644 ms
• Initialisation	719 ms	713 ms	758 ms	755 ms	785 ms	804 ms
• Exploration	358 ms	701 ms	1936 ms	290 ms	756 ms	1840 ms
POR disabled	2060 ms	13894 ms	174620 ms	2060 ms	13894 ms	174620 ms
Reduction	*48%*	*90%*	*98%*	*49%*	*89%*	*98%*
Memory consump.						
POR enabled	12 MB	17 MB	24 MB	12 MB	17 MB	24 MB
POR disabled	23 MB	24 MB	38 MB	23 MB	24 MB	38 MB
Reduction	*48%*	*29%*	*37%*	*48%*	*29%*	*37%*

Table 7.2: Resource consumption during verification of `buggyBlenderAgent` with respect to the property $\varphi = \mathsf{F}\,\mathbf{bel}(\texttt{filled})$, and in which i is the number of bananas and oranges that need be added to the blender's reservoir.

Property φ' states that always if `buggyBlenderAgent` believes that it has ticked the check box indicating that sufficiently many bananas have been added, indeed this is the case; φ'' states the equivalent for oranges. It turns out that `buggyBlenderAgent` violates both these properties. For example, with respect to φ', the counterexample that is generated in all cases is one on which `buggyBlenderAgent` first adds $i-1$ bananas, subsequently adds i oranges, and finally ticks the bananas check box on the recipe, even though one banana too few has been added. We verified both properties with and without POR enabled; the verification statistics for φ' are given in Table 7.3: the reductions are substantial in most cases, although overhead in the $i = 4$ case outweighs the benefits of the reduction (but in the absolute sense, the differences are negligible). Verifying φ'' with POR enabled did not result in any reduction, because the first computation investigated by the model checker already violates φ''. Thus, in this case, there was no room for improvements.

Again, no significant differences can be observed between the use of $\mathrm{H}^{sing}_{\mathrm{POR}}$ and $\mathrm{H}^{mult}_{\mathrm{POR}}$. In the next subsection, we have a look at a second case study in which $\mathrm{H}^{sing}_{\mathrm{POR}}$ will not yield any reduction, while $\mathrm{H}^{mult}_{\mathrm{POR}}$ does.

7.3.2 `blocksCounterAgent`

To illustrate the potential of $\mathrm{H}^{mult}_{\mathrm{POR}}$, we reconsider `blocksCounterAgent`. Recall from Sect. 6.4.2 that `blocksCounterAgent`, whose source code appears in Fig. 6.3, is a merger of the agents `blocksAgent` and `counterAgent` (introduced in Sect. 4.3). Suppose we want to verify whether `blocksCounterAgent` eventually believes to have moved all blocks to the table. We can express this in LTL as $\varphi = \mathsf{F}\neg$**bel**`(on(X,Y),not(Y=table))`. Table 7.4 shows the verification statistics when model checking `blocksCounterAgent` with respect to φ for various values of i in `target(`i`)` (i.e. the upper limit on the numbers to which `blocksCounterAgent` can count; see also Sect. 6.4.2).

In this example, there are clear differences between the reduction obtained when using $\mathrm{H}^{sing}_{\mathrm{POR}}$ and $\mathrm{H}^{mult}_{\mathrm{POR}}$: while use of the former does not reduce the transition system at all, $\mathrm{H}^{mult}_{\mathrm{POR}}$ yields reductions in the 55%-70% range. Because $\mathrm{H}^{sing}_{\mathrm{POR}}$ does not reduce the transition system, while

	Heuristic $\mathrm{H}^{sing}_{\mathrm{POR}}$ for **C1**			Heuristic $\mathrm{H}^{mult}_{\mathrm{POR}}$ for **C1**		
	$i = 4$	$i = 16$	$i = 64$	$i = 4$	$i = 16$	$i = 64$
States						
POR enabled	14	38	134	14	38	134
POR disabled	34	118	454	34	118	454
Reduction	*59%*	*68%*	*70%*	*59%*	*68%*	*70%*
Verification time						
POR enabled	1049 ms	1524 ms	2676 ms	1118 ms	1443 ms	2840 ms
• Initialisation	730 ms	715 ms	808 ms	834 ms	706 ms	761 ms
• Exploration	319 ms	809 ms	1868 ms	284 ms	747 ms	2079 ms
POR disabled	944 ms	2183 ms	5869 ms	944 ms	2183 ms	5869 ms
Reduction	*-10%*	*30%*	*54%*	*-16%*	*34%*	*52%*
Memory consump.						
POR enabled	12 MB	17 MB	24 MB	12 MB	17 MB	24 MB
POR disabled	12 MB	24 MB	26 MB	12 MB	24 MB	26 MB
Reduction	*0%*	*29%*	*8%*	*0%*	*29%*	*8%*

Table 7.3: Resource consumption during verification of `buggyBlenderAgent` with respect to the property $\varphi' = \mathsf{G}$[**bel**`(ticked(bananas))` $\rightarrow$ **bel**`(recipe(bananas,Qr),added(bananas,Qr))`], and in which i is the number of bananas and oranges that need be added to the blender's reservoir.

it does require additional computations that are not carried out when Por is disabled, resource consumption is higher during verification with $\mathrm{H}^{sing}_{\mathrm{POR}}$ than without Por. Note however, that the overhead of using $\mathrm{H}^{sing}_{\mathrm{POR}}$ can be ascribed entirely to the initialisation phase of the Por algorithm: the exploration time when Por is enabled combined with $\mathrm{H}^{sing}_{\mathrm{POR}}$ is roughly the same as when Por is disabled.

To understand why $\mathrm{H}^{mult}_{\mathrm{POR}}$ is able to reduce the transition system in contrast to $\mathrm{H}^{sing}_{\mathrm{POR}}$, let us have a closer look at the dependence graph for the transition classes of `blocksCounterAgent` in Fig. 7.5b. Recall that labels on vertices (i.e. transition classes) correspond to lines in `blocksCounterAgent`'s source code in Fig. 6.3 on which action rules are defined. Now, when using $\mathrm{H}^{sing}_{\mathrm{POR}}$, every candidate ample sets consists of transitions belonging to the same transition class. Unfortunately, the transitions in the 12-13 transition class are visible to φ (i.e. moving a block to the table may cause φ to be satisfied), and the transitions in the 15-16 and 18-19 transition classes are dependent on transitions belonging to another transition class (namely 18-19 and 15-16, respectively) as demonstrated by the edge between these two transition classes in Fig. 7.5b. Consequently, with $\mathrm{H}^{sing}_{\mathrm{POR}}$, no ample sets are ever found.

Next, let us consider $\mathrm{H}^{mult}_{\mathrm{POR}}$. Recall that $\mathrm{H}^{mult}_{\mathrm{POR}}$ is a generalisation of $\mathrm{H}^{sing}_{\mathrm{POR}}$ that searches for connected components in the dependence graph. The dependence graph has two such components: $\boldsymbol{T}_1 = \{12\text{-}13\}$ and $\boldsymbol{T}_2 = \{15\text{-}16, 18\text{-}19\}$. The transitions generated by transition class 12-13 in the former connected component are visible to φ as aforementioned such that $\boldsymbol{T}_1$ never yields an ample set. The latter connected component, however, does generate ample sets. To see this, note that there exist no transition classes outside $\boldsymbol{T}_2$ with which a transition class in $\boldsymbol{T}_2$ is dependent. Also, there exist no transition classes outside $\boldsymbol{T}_2$ that can enable a transition class in $\boldsymbol{T}_2$ (the influence graphs in Fig. 6.4 demonstrate that transitions belonging to transition class 12-13 cannot enable a transition belonging to transition classes 15-16 and 18-19). Thus, the enabled transitions belonging to transition classes in $\boldsymbol{T}_2$ always constitute an ample set. This accounts for the reduction that is obtained when $\mathrm{H}^{mult}_{\mathrm{POR}}$ is used. In the next subsection, we show that $\mathrm{H}^{mult}_{\mathrm{POR}}$ yields at least the same reduction as $\mathrm{H}^{sing}_{\mathrm{POR}}$.

	Heuristic $\mathrm{H}^{sing}_{\mathrm{POR}}$ for **C1**			Heuristic $\mathrm{H}^{mult}_{\mathrm{POR}}$ for **C1**		
	$i = 4$	$i = 16$	$i = 64$	$i = 4$	$i = 16$	$i = 64$
States						
Por enabled	27	99	387	11	35	131
Por disabled	27	99	387	27	99	387
Reduction	*0%*	*0%*	*0%*	*59%*	*65%*	*66%*
Verification time						
Por enabled	1170 ms	2366 ms	5884 ms	551 ms	1218 ms	2322 ms
• Initialisation	256 ms	238 ms	214 ms	216 ms	225 ms	232 ms
• Exploration	914 ms	2128 ms	5670 ms	335 ms	993 ms	2090 ms
Por disabled	857 ms	2158 ms	5789 ms	857 ms	2158 ms	5789 ms
Reduction	*-27%*	*-9%*	*-2%*	*36%*	*44%*	*60%*
Memory consump.						
Por enabled	12 MB	25 MB	25 MB	6 MB	12 MB	20 MB
Por disabled	12 MB	24 MB	27 MB	12 MB	24 MB	27 MB
Reduction	*0%*	*-4%*	*7%*	*50%*	*50%*	*26%*

Table 7.4: Resource consumption during verification of `blocksCounterAgent` with respect to the property $\varphi = \mathsf{F}\neg$**bel**`(on(X,Y),not(Y=table))`, and in which i is the upper limit on the numbers to which `blocksCounterAgent` can count.

7.3.3 Discussion

From the previous two case studies, we have learned several things about our POR implementation for GOAL. We discuss these here. For one thing, POR is able to reduce the transition system significantly in some of the cases. However, there are also instances in which running the POR algorithm has no effects on the transition system. In such cases, due to the POR algorithm's initialisation phase, resource consumption is higher than when POR is disabled. We identify two situations in which the POR algorithm is able to come to good results.

1. *Independent tasks*

   If a GOAL agent needs to carry out independent tasks, then reductions can be expected. However, if the action rules of an agent need be applied in some sequential order (like we saw in the first part of the `blenderAgent` example), significant reductions are unlikely. This is not a shortcoming of the GOAL implementation of POR, but rather inherent to the method in general. In [23], it is stated that "concurrent systems, and in particular distributed programs, which exhibit a lot of parallelism and independency, are the main focus of the partial order reduction". Similarly, in [96], it is stated that "the stubborn set method [based on the same principles as the ample set method] usually does not give good reduction results and is thus not worth using when there is no concurrency". We do speculate, however, that GOAL agents typically are tighter coupled than multi-process systems. As such, our POR implementation may be effective in practice less frequently than POR implementations for concurrent systems.

2. *Over-underspecification*

   Although our POR implementation may be effective less often in practice than POR implementations for concurrent systems, we see a specific application of POR combined with GOAL that is not present when model checking imperative programs: the identification of undesired behaviour due to too much underspecification. We showed an example of this in the second part of the `blenderAgent` case study. We are not aware of publications identifying similar connections between certain types of bugs in programs written in an imperative language (e.g. PROMELA) and the effectiveness of partial order reduction when verifying such programs.

A question unanswered upto now is which heuristic we should choose. As often, this depends on the situation. Because $\mathrm{H}^{sing}_{\mathrm{POR}}$ considers transition classes in isolation, the initialisation phase of the POR algorithm with $\mathrm{H}^{sing}_{\mathrm{POR}}$ consists of a single pass through all transition classes to determine whether there exists a transition class that is independent of all others and cannot be enabled by other transitions classes. That is, $\mathrm{H}^{sing}_{\mathrm{POR}}$ requires time linear in the number of transition classes n, i.e. $O(n)$. In contrast, the initialisation phase of the algorithm when using $\mathrm{H}^{mult}_{\mathrm{POR}}$ requires a depth-first traversal of the graph corresponding to the ApproxDep relation. Let m be the number of elements in ApproxDep. Then, $\mathrm{H}^{mult}_{\mathrm{POR}}$ requires time linear in $n+m$ due to the depth-first search algorithm, i.e. $O(n+m)$. Thus, $\mathrm{H}^{sing}_{\mathrm{POR}}$ is slightly easier to compute than $\mathrm{H}^{mult}_{\mathrm{POR}}$. If there are few dependences (i.e. m is small), the differences will be slim, but if there are many dependences (i.e. m is large), the differences may be larger. Moreover, if there are many dependences, i.e. the system is tightly coupled, the POR algorithm is unlikely to yield a reduction as argued above. Thus, in such cases, the "$+m$" cuts both ways in the negative sense: it imposes more overhead, while the likelihood of reduction is small.

However, although $\mathrm{H}^{sing}_{\mathrm{POR}}$ is easier to compute, $\mathrm{H}^{mult}_{\mathrm{POR}}$ is able to yield ample sets when $\mathrm{H}^{sing}_{\mathrm{POR}}$ is not. We saw this in the `blocksCounterAgent` case study. More generally, $\mathrm{H}^{mult}_{\mathrm{POR}}$ finds any ample set that $\mathrm{H}^{sing}_{\mathrm{POR}}$ would find. To see this, note that a transition class that would generate a candidate ample set using $\mathrm{H}^{sing}_{\mathrm{POR}}$ is a singleton connected component in the graph corresponding to ApproxDep. Thus, when using $\mathrm{H}^{mult}_{\mathrm{POR}}$, we know that we at least obtain the reduction that we would have gained by using $\mathrm{H}^{sing}_{\mathrm{POR}}$.

In any case, there is a trade-off between the two heuristics, and it is difficult to designate either $\mathrm{H}^{sing}_{\mathrm{POR}}$ or $\mathrm{H}^{mult}_{\mathrm{POR}}$ as best. Because computational overhead depends on the number of transition classes and the size of ApproxDep, one rule of thumb is that in the presence of many action rules (hence many transition classes) and dependences, a first verification attempt should be run with $\mathrm{H}^{sing}_{\mathrm{POR}}$. If this does not yield a reduction (this can already be established after the initialisation phase), a second run can be tried with $\mathrm{H}^{mult}_{\mathrm{POR}}$.

7.4 Complementary Techniques

We have applied Por in a single-agent setting. A Por algorithm for multi-agent Goal systems is complementary to the work presented here, and would be more similar to settings in which Por traditionally is applied: the reduction of different interleavings of transitions executed by different agents. We believe it would be most effective to run several Por algorithms in parallel: one for each agent based on the work in this chapter, and a separate Por algorithm for tracking interaction (communication or mutations to the environment) between agents and reducing the different interleavings. The latter involves the definition of a new dependence relation, say AgentDep, at the level of agents rather than at the level of transition classes. For instance, suppose we have a multi-agent Goal system in which agent P sends messages to agent P'. Then, we might have $\mathsf{AgentDep}(P, P')$. The partial order reduction algorithm for tracking agent interaction subsequently should be defined in terms of AgentDep rather than ApproxDep as we did in this chapter. As such, we would have both the benefits of single-agent Por and Por as traditionally applied in concurrent settings, while maintaining a high degree of modularity.

Additionally, the Por implementation presented in this chapter is complementary to the Pbs algorithm introduced in the previous chapter. More specifically, these two methods can be combined as follows. First the Pbs algorithm is run to slice the agent. Subsequently, the Por algorithm is initialised for either $\mathrm{H}^{sing}_{\mathrm{POR}}$ or $\mathrm{H}^{mult}_{\mathrm{POR}}$, and finally, the actual verification procedure is started (with the Por algorithm enabled). By using this order, the Por algorithm's initialisation phase also benefits from the Pbs algorithm, as the Pbs algorithm may already have sliced away some of the transition classes, which subsequently need not be considered during initialisation of Por. Note that if we would reverse the order, i.e. first initialise Por and then run the Pbs algorithm, such benefits are lost.

Because both the Pbs and Por algorithms are defined in terms of the same relations on transitions (visibility, enabled-by, independence), their implementations share the same code base for a large part. This is an advantage of using the transition theory presented in Chap. 5 as a basis for both these techniques from a software engineering point of view. Additionally, when the two methods are both used, results of computations carried out during execution of the Pbs algorithm (which should be run first as outlined above) can be reused during initialisation and execution of the Por algorithm. This is a benefit from a computational point of view. As such, we have adopted a recent recommendation found in the 2009 publication [78] that researchers not only should focus on developing new reduction techniques, but also on how existing techniques can be combined (specifically, in [78]'s conclusion, the following question is raised: "can information gathered by one technique be used by another technique?").

7.5 Summary

In this chapter, we have described a partial order reduction algorithm for Goal agents. We started with a brief introduction to Por methods, and have treated the ample set method in more detail. Next, we have discussed two heuristics for computing ample sets tailored to Goal, although one of them has strong conceptual similarities with the heuristic used in Spin. In two case studies, we have shown in which instances our Por implementation can be advantageous. We have argued that Por for Goal agents may yield a reduction less often than when Por is applied to concurrent systems, because Goal agents seem tighter coupled than multi-process programs. However, we have also identified a situation in which Por combined with Goal can

reduce the transition system that is not present when concurrent systems are under investigation: undesired behaviour due to too much underspecification. Finally, we have treated how POR can be combined with property-based slicing, and argued that basing both techniques on the transition theory presented in Chap. 5 is beneficial. The work presented in this chapter is novel in at least two ways. First, to the best of our knowledge, no POR algorithm has been implemented for the verification of agents written in an actual agent programming language. Second, POR methods in the model checking literature are applied to reduce the number of interleavings of concurrent events generated by different processes; we have considered POR in the context of a single process, i.e. a single GOAL agent.

Chapter 8

Epilogue

In this final chapter, we summarise the previous in Sect. 8.1, draw conclusions in Sect. 8.2, and give an overview of possible directions for future work in Sect. 8.3.

8.1 Summary

Chapter 1 introduced the subject of the thesis, its motivation and scope, the main contributions, and an outline of the remainder.

In Chap. 2, we have treated the GOAL agent language. We started with an exposition on how knowledge, beliefs, and goals are represented, continued with the notion of mental states, proceeded with a treatise of actions, and ended with GOAL's semantics.

In Chap. 3, we have discussed the theory and implementation of a novel model checker for GOAL that is built on top of the standard interpreter. In the first two sections, we have presented a verification logic for GOAL based on LTL, and briefly discussed the theory of model checking and considerations for choosing particular algorithms and approaches. With respect to implementation, we have presented the architecture of the implemented system: a model checking framework for any language with an operational semantics, in this thesis instantiated for GOAL. Additionally, we discussed the interaction between this framework and the GOAL interpreter. A keystone of this interaction is the conversion of mental states to their binary representation and back.

In Chap. 4, we have compared our interpreter-based model checker for GOAL with other efforts in agent verification. We started with an overview of related work in the field of APL model checking. During this exposition, we listed three benefits of our approach over existing approaches: improved performance, increased expressiveness of properties, and immediate / encoding-less language support. The remainder of the chapter has focussed on the first of these claimed advantageous with a quantitative performance analysis of our model checker and two other model checkers for GOAL. The results indeed show that our approach is a lot more efficient. Along the way, we have proposed regression analysis as a statistical tool to assess scalability of model checkers as well as criteria that state when a model checker scales well in experimental conditions.

In Chap. 5, we introduced a transition theory for GOAL according to the POR notion of transitions: one in which transitions can be thought of as operations on mental states rather than as mental state pairs in the transition system. The first section of the chapter elaborated on this concept and its formal definition. The two sections that followed concerned read and write sets. Finally, in the last section, we defined several relations on transitions that are used by the POR and PBS algorithms presented in later chapters. Central to most of the chapter was the observation that we can only approximate write sets of transitions (and the relations defined in terms of them) because at the time that we want to compute them, typically before the transition system is generated, we have insufficient information for a precise computation.

In Chap. 6, we have investigated property-based slicing for GOAL agents. After outlining the idea of PBS (with respect to model checking) in the first section, we have treated a graph-theoretic and relational account of how transitions influence each other and the property under investigation.

The two formalisms were proven to correspond with each other, enabling the formulation of the algorithm in terms of the influence graph, while proving this algorithm correct using the influence relation / influential set. Preliminary case studies with an implementation of the algorithm show that slicing has the potential to yield substantial reductions, specifically in case the agent under verification has subtasks that neither influence each other, nor the property; we have argued that such subtasks correspond to SCCs in the influence graph. Finally, some extensions to the basic algorithm were proposed, which have not yet been implemented.

In Chap. 7, we have described a partial order reduction algorithm for Goal agents. We started with a brief introduction to Por methods, and have treated the ample set method in more detail. Subsequently, we discussed two heuristics for computing ample sets tailored to Goal, although one of them has strong conceptual similarities with the heuristic used in Spin. In two case studies, we have shown in which instances our Por implementation can be advantageous. We have argued that Por for Goal agents may yield a reduction less often than when Por is applied to concurrent systems, because Goal agents seem tighter coupled than multi-process programs. However, we have also identified a situation in which Por combined with Goal can reduce the transition system that is not present when concurrent systems are under investigation: undesired behaviour due to too much underspecification. Finally, we have treated how Por can be combined with property-based slicing, and argued that basing both techniques on the transition theory presented in Chap. 5 is beneficial.

Finally, this chapter, i.e. Chap. 8, concludes the thesis with a summary, conclusion, and future work.

8.2 Conclusion

The primary motivation for the work presented in this thesis was to test the hypothesis that reusing existing language interpreters to build model checkers is a better approach to agent program verification than reusing existing model checkers. In this respect, Chap. 4 is a vital chapter. Not only did we show that our approach performs substantially better than existing tools for Goal verification, we identified two additional benefits of interpreter-based model checkers: increased expressiveness of the verification logic and immediate / encoding-less language support.

The second part of the thesis has focussed on further optimising the core model checker by extending it with state space reduction techniques. Preliminary case studies have shown that these techniques, although conceived for imperative language model checkers, can be successfully applied in agent verification as well, if certain conditions are met.

Finally, we stress that by no means we want to suggest that we have *solved* agent program model checking. We merely have approached the problem from a different angle; an angle more effective than those of the past.

8.3 Future Work

There is still much work to be done in this exciting field of research. Below, we list some of the possible directions for future work.

Extensions

- *Multi-agent support*
  Although we have focused on single-agent systems in this thesis, our primary interest is of course the verification of multi-agent systems. Due to modularity of our Java framework, adding multi-agent support to the current model checker does not require fundamental changes to the architecture.
- *Environment support*
  Environment support seems not much different from multi-agent support: the environment can simply be regarded as an agent. In practice, however, Goal uses the *Environment Interface Standard* proposed in [4] for the implementation of environments,

which is JAVA-based and not designed with model checking in mind. Incorporating such environments in the model checker in a neat fashion may pose challenges. Verification of *open systems* that interact with an environment is called *module checking* in the literature (e.g. [65, 66]).

- *Generalised model checking*
  With the current model checker, we can establish that a GOAL agent is correct given its initial mental state. Such results give us *confidence* in the fact that the agent is also correct if the initial mental state is different but comparable. However, such confidence may not be sufficient when safety-critical applications are under investigation. Rather than model checking an agent for each relevant initial mental state in such case, techniques might be developed to generalise the results of one verification run to other instances (perhaps when certain conditions are met).

Optimisations

- *Experimentation with* PBS *and* POR
  To get a better feel for when PBS and POR reduce the state space in practice, more experience with the methods is needed.
- *Binary mental state compression*
  If an agent can have many different beliefs (or goals) during a verification run of which only a small portion is actually believed in each mental state (e.g. `counterAgent`, which has an infinite number of beliefs about what its current number can be, but only one such belief at the same time), a binary mental state consists mostly of 0-bits. There might be more efficient ways to represent such sequences by means of compression, e.g. using Huffman encoding [62] or by incorporating ideas of the collapse method (e.g. [56]).
- *Symbolic model checking*
  It has occurred to us that although symbolic methods based on binary decision diagrams are traditionally applied to hardware model checking, we might be able to benefit from them as well. One of the reasons for symbolic model checking to work well on hardware specifications is that boolean data (i.e. bit strings), which are well represented using binary decision diagrams, are common (e.g. [58]). Although such data occur less frequently in imperative software, making symbolic model checking less attractive, binary mental states are exactly such structures. Hence, benefits may be gained.
- *Multi-core model checking*
  The current implementation of the model checker is run as a single-threaded JAVA program. As the increase in processor speed seems to have levelled, while the number of cores per processor seems to grow, benefits may be gained if these multiple cores can be used during model checking. Over the last couple of years, a movement within the model checking community has emerged that focusses on tailoring model checking algorithms to multi-core execution [15]. Apart from the incorporation of such algorithms in our framework, we also see room for GOAL-specific optimisations in this respect: the distribution of successor generation. We have found (in an unstructured and informal series of test runs) that roughly 80% of the time of a verification run is spent purely on the actual generation of successors, i.e. the process in which the interpreter computes a successor given a mental state and action. Such computations can be carried out in parallel, which should result in a decrease of verification time.

Other

- *Comparison with existing model checkers*
  In this thesis, we have compared our interpreter-based model checker with other model checkers for GOAL. We think it would also be interesting to see how our approach, and

agent model checking in general, compares to existing state-of-the-art model checkers like Spin. Specifically, it would be interesting to investigate whether there are problems that are more easily implemented and verified with our Goal model checker than with Spin. If this were to be the case, which we speculate it is, not only would this be a point in favour of our model checker, but also for agent-oriented programming as a programming paradigm for agents in general.

- *Program component for other agent languages*

  Developing program components for other agent languages like AgentSpeak is interesting for several reasons. With respect to AgentSpeak, it offers a way to validate our findings regarding the efficiency of an interpreter-based approach, because AgentSpeak model checkers based on Spin and Jpf exist, as well as an Ail implementation of the interpreter. Consequently, the experiments presented in Chap. 4 can be repeated for AgentSpeak. Also, having program components for more than one agent language combined with multi-agent support opens the door to verification of *heterogeneous* systems (similar to [31]). Another interesting option is to connect our framework to the Ail interpreter.

- *Meta model checking and Theory of Mind*

  This final possible direction for future work is not related to the implementation of our current system, but to model checking Goal in general. We believe it would be very interesting to extend Goal with an implementation of the Theory of Mind, i.e. enabling Goal agents to have mental models of other agents. Given such an implementation, it *may* be possible to develop a Goal agent that can model check another Goal agent based on the beliefs it has about it. We do not think that such a meta model checker would perform better than our current, but from a theoretical point of view it would be quite an achievement.

Part III

Appendices

Appendix A

Büchi Automata

In this appendix, we treat in more detail Büchi automata as presented in Sect. 3.2.1. Recall from that section that a Büchi automaton $\mathcal{A}$ is a tuple $\mathcal{A} = \langle \mathcal{Q}, \mathcal{I}, \delta, \mathcal{F}, \mathcal{D}, \Lambda \rangle$, in which (e.g. [89]):

- $\mathcal{Q}$ is a finite set of *states*,
- $\mathcal{I} \subseteq \mathcal{Q}$ is the set of *initial states*,
- $\delta : \mathcal{Q} \longrightarrow 2^{\mathcal{Q}}$ is the *transition function*,
- $\mathcal{F} \subseteq 2^{\mathcal{Q}}$ is a set of *acceptance conditions* such that each acceptance condition $F \subseteq \mathcal{Q}$,
- $\mathcal{D}$ is a set of *labels*, called the *domain*, and
- $\Lambda : \mathcal{Q} \longrightarrow 2^{\mathcal{D}}$ is a *labelling function*.

In the following, we treat the definition of program automata (Appx. A.1), property automata (Appx. A.2), product automata (Appx. A.3), and the check for language emptiness (Appx. A.4).

A.1 Program automata

Recall from Sect. 2.5 that the semantics of a GOAL agent is defined by its transition system $\mathcal{T} = \langle \Omega_M, \mu_0, \longrightarrow \rangle$. To define the program automaton for a GOAL agent P, we first extend $\mathcal{T}$ to a *Kripke structure*. A Kripke structure (e.g. [24]) is a transition system in which each state is labelled with its *valuation*, i.e. the set of atomic propositions that are true in that state. Recall from Sect. 3.1.1 that for GOAL the set of atomic propositions is the language of mental state conditions $\mathcal{L}_\Psi$. To find the atomic propositions, i.e MSCs, that are true in a mental state μ, we define a function $\mathcal{V} : \Omega_M \rightarrow 2^{\mathcal{L}_\Psi}$, called the *valuation function* (e.g. [76]), that maps each mental state μ to the MSCs that are true in μ, i.e. $\mathcal{V}(\mu) = \{ \psi \in \mathcal{L}_\Psi \mid \mu \models_\Psi \psi \}$. Subsequently, the Kripke structure of P is the tuple $\langle \mathcal{T}, \mathcal{V} \rangle$ (alternatively, e.g. in [24], the Kripke structure is defined by adding $\mathcal{V}$ to $\mathcal{T}$, i.e. as the tuple $\langle \Omega_M, \mu_0, \longrightarrow, \mathcal{V} \rangle$). Given the Kripke structure of P, its corresponding automaton $\mathcal{A}_P = \langle \mathcal{Q}_P, \mathcal{I}_P, \delta_P, \mathcal{F}_P, \mathcal{D}_P, \Lambda_P \rangle$ is defined as follows (derived from [93]).

$$
\begin{array}{lcl}
\mathcal{Q}_P & = & \Omega_M \cup \{ \imath_P \} \\
\mathcal{I}_P & = & \{ \imath_P \} \\
\delta_P(q_P) & = & \begin{cases} \{ \mu' \in \Omega_M \mid \langle q_P, \mu' \rangle \in \longrightarrow \} & \text{if } q_P \in \Omega_M \\ \mu_0 & \text{if } q_P = \imath_P \end{cases} \\
\mathcal{F}_P & = & \{ \Omega_M \cup \{ \imath_P \} \} \\
\mathcal{D}_P & = & 2^{\mathcal{L}_\Psi} \\
\Lambda_P(q_P) & = & \begin{cases} \{ \mathcal{V}(q_P) \} & \text{if } q_P \in \Omega_M \\ \mathcal{D}_P & \text{if } q_P = \imath_P \end{cases}
\end{array}
$$

That is, the states of $\mathcal{A}_P$ are all the mental states that P can be in, extended with a *source state* $\imath_P$, similar to the source state in the example automaton in Fig. 3.1. The transition function is the same as the transition relation $\longrightarrow$, extended with a transition from $\imath_P$ to μ_0, i.e. the actual initial state of the agent. The set of acceptance conditions is a singleton containing all states of the automaton such that every run of $\mathcal{A}_P$ is accepting. To see this, observe that because $\mathcal{Q}_P$ is a finite set, every infinite run must visit at least one $q_P \in \mathcal{Q}_P$ infinitely often. The domain is the power set of all mental state conditions $\mathcal{L}_\Psi$, such that each label is a set of MSCs. Finally, the labelling function labels each state with a single label containing the MSCs that are true in it according to the valuation function $\mathcal{V}$; the source state is trivially labelled with all possible sets of MSCs, i.e. the entire domain.

The definition here is not identical to the definition in [93], because in that paper, labels are on transitions rather than on states: the label on a transition is the valuation of the state from which the transition leaves. In our definition, we associate this label with that state itself. Also, in [93], no source state is used. We introduce this source state to enable a more straightforward definition of the product automaton. This is also done in [24].

As an example, the structure of the Büchi automaton for `blenderAgent` is given in Fig. A.1. The similarities with its transition system (given in Fig. 2.2) are apparent such that a run of this automaton corresponds to a computation of `blenderAgent` in a straightforward manner. What does the labelling function look like, and which words does the automaton accept? To answer these questions, we first need to fix the domain, i.e. the contents of $\mathcal{L}_\Psi$. Theoretically, $\mathcal{L}_\Psi$ is infinite, which is quite inconvenient from a computational point of view. Therefore, in practice, we define the domain as a *sufficiently large* finite subset of $\mathcal{L}_\Psi$ based on the analysis of the property automaton. In the next section, we will see that a "sufficiently large" subset of $\mathcal{L}_\Psi$ is the subset containing all the mental state conditions that occur in the negated property $\neg\varphi$ under investigation. For example, suppose that the negated property that we want `blenderAgent` to satisfy is $\neg\varphi = \mathsf{G}\neg\mathbf{bel}(\texttt{ticked(bananas)})$. Then, a sufficiently large subset is $\{\mathbf{bel}(\texttt{ticked(bananas)})\}$; in the remainder of this example, we assume $\mathcal{L}_\Psi = \{\mathbf{bel}(\texttt{ticked(bananas)})\}$ such that the domain is $\mathcal{D}_P = \{\emptyset, \{\mathbf{bel}(\texttt{ticked(bananas)})\}\}$.

In which mental states is $\mathbf{bel}(\texttt{ticked(bananas)})$ true? Well, from the belief bases given in Fig. 2.2, we can derive that this is the case in mental states μ_4, μ_5, μ_6, μ_7, and μ_8. Consequently, $\mathcal{V}(\mu_i) = \{\mathbf{bel}(\texttt{ticked(bananas)})\}$ for $4 \leq i \leq 8$, and $\mathcal{V}(\mu_i) = \emptyset$ for $0 \leq i \leq 3$ and $9 \leq i \leq 17$. Then, the labelling function is defined as follows.

$$\Lambda_P(\mu_i) \quad = \quad \begin{cases} \{\emptyset\} & \text{if } 0 \leq i \leq 3 \text{ or } 9 \leq i \leq 17 \\ \{\{\mathbf{bel}(\texttt{ticked(bananas)})\}\} & \text{if } 4 \leq i \leq 8 \\ \{\emptyset, \{\mathbf{bel}(\texttt{ticked(bananas)})\}\} & \text{if } q_P = \imath_P \end{cases}$$

Let $\Psi = \{\mathbf{bel}(\texttt{ticked(bananas)})\}$. Below, we give some examples of accepting runs and words.

$$\begin{array}{lcl@{\qquad}lcl} r_1 & = & \imath_P\mu_0\mu_1\mu_2\mu_3\mu_4\mu_5\mu_6\mu_7\mu_8\mu_8\cdots & r_2 & = & \imath_P\mu_0\mu_1\mu_{15}\mu_{12}\mu_9\mu_{10}\mu_6\mu_7\mu_8\mu_8\cdots \\ \omega_1 & = & \emptyset\ \emptyset\ \emptyset\ \emptyset\ \emptyset\ \Psi\ \Psi\ \Psi\ \Psi\ \Psi\ \Psi\cdots & \omega_2 & = & \emptyset\ \emptyset\ \emptyset\ \emptyset\ \emptyset\ \emptyset\ \emptyset\ \Psi\ \Psi\ \Psi\ \Psi\cdots \\ \omega_1' & = & \Psi\ \emptyset\ \emptyset\ \emptyset\ \emptyset\ \Psi\ \Psi\ \Psi\ \Psi\ \Psi\ \Psi\cdots & \omega_2' & = & \Psi\ \emptyset\ \emptyset\ \emptyset\ \emptyset\ \emptyset\ \emptyset\ \Psi\ \Psi\ \Psi\ \Psi\cdots \end{array}$$

In contrast, the words $\omega_3 = \Psi\Psi\cdots$ (i.e. an infinite sequence of Ψs) is not in the automaton's language, because there exists no accepting run whose all states are labelled with Ψ. Similarly, $\omega_4 = \Psi\emptyset\Psi\emptyset\cdots$ (i.e. an infinite sequence containing Ψ and $\emptyset$ in alternating order) is not an accepted word.

A.2 Property automata

The construction of the property automaton corresponding to some LTL formula ϕ is less straightforward than the program automaton. As remarked in Sect. 3.2.1, various translation algorithms exist. Here, we discuss [29]'s LTL2Aut algorithm (we refer to Sect. 3.2.1 for a motivation of our choice for LTL2Aut). The heart of this algorithm comprises the computation of *covers*. The cover

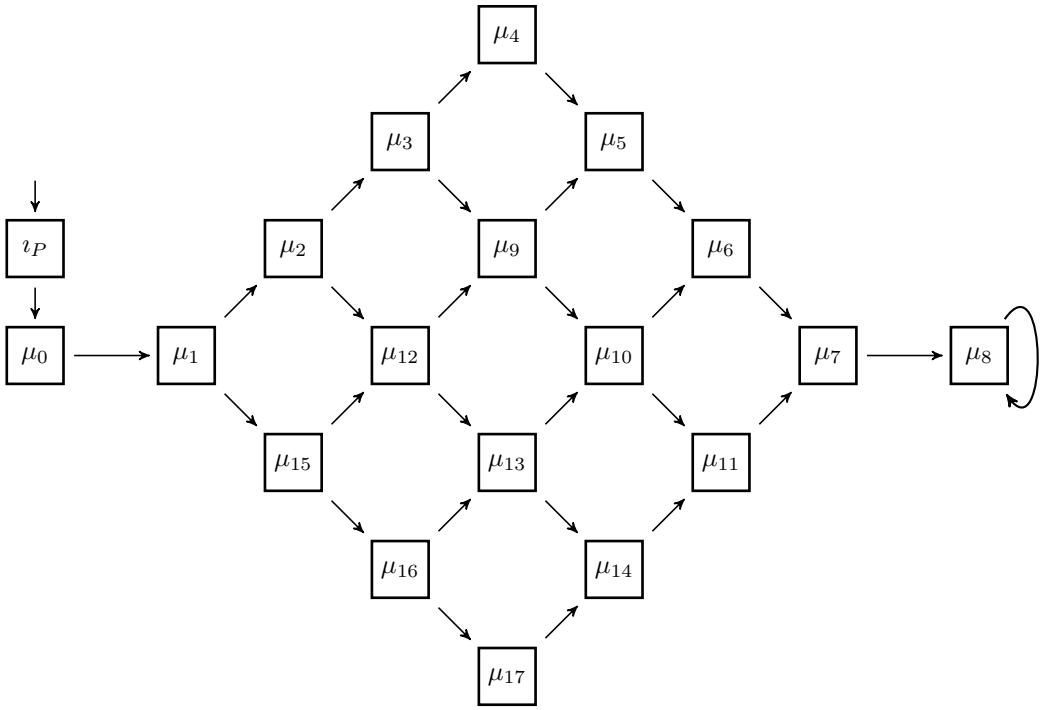


Figure A.1: Büchi automaton for `blenderAgent`.

of a set of LTL formulas Φ, denoted $cover(\Phi) \subseteq 2^{\mathcal{L}_{\text{LTL}}}$, contains sets $C \subseteq \mathcal{L}_{\text{LTL}}$ of LTL formulas $\phi' \in \mathcal{L}_{\text{LTL}}$ such that the following equivalence is true [29]:

$$\bigwedge_{\phi \in \Phi} \phi \quad \equiv \quad \bigvee_{C \in cover(\Phi)} \left[\bigwedge_{\phi' \in C} \phi' \right]$$

The cover of a set of LTL formulas may be regarded as the LTL equivalent to what is called *disjunctive normal form* in propositional logic (e.g. [18]). To compute the cover of a set Φ, the *non-elementary formulas* in Φ are *decomposed* into *elementary formulas*. An elementary formula is a X formula, a proposition, or the negation of a proposition; a non-elementary formula is every formula that is not elementary. The decomposition of a non-elementary formula ϕ is defined as two sets, denoted $elem_1(\phi)$ and $elem_2(\phi)$, such that the $\phi = \left[\bigwedge_{\phi' \in elem_1(\phi)} \phi'\right] \vee \left[\bigwedge_{\phi' \in elem_2(\phi)} \phi'\right]$. The sets $elem_1(\phi)$ and $elem_2(\phi)$ are defined as follows (the definition is identical to [29], extended with cases for F and G for notational convenience in the remainder):

$$elem_1(\phi) = \begin{cases} \{\phi_1, \phi_2\} & \text{if } \phi = \phi_1 \wedge \phi_2 \\ \{\phi_1\} & \text{if } \phi = \phi_1 \vee \phi_2 \\ \{\phi'\} & \text{if } \phi = \mathsf{F}\phi' \\ \{\bot\} & \text{if } \phi = \mathsf{G}\phi' \\ \{\phi_2\} & \text{if } \phi = \phi_1 \,\mathsf{U}\, \phi_2 \\ \{\phi_1, \phi_2\} & \text{if } \phi = \phi_1 \,\mathsf{R}\, \phi_2 \end{cases} \qquad elem_2(\phi) = \begin{cases} \{\top\} & \text{if } \phi = \phi_1 \wedge \phi_2 \\ \{\phi_2\} & \text{if } \phi = \phi_1 \vee \phi_2 \\ \{\mathsf{X}\mathsf{F}\phi'\} & \text{if } \phi = \mathsf{F}\phi' \\ \{\phi', \mathsf{X}\mathsf{G}\phi'\} & \text{if } \phi = \mathsf{G}\phi' \\ \{\phi_1, \mathsf{X}[\phi_1 \,\mathsf{U}\, \phi_2]\} & \text{if } \phi = \phi_1 \,\mathsf{U}\, \phi_2 \\ \{\phi_2, \mathsf{X}[\phi_1 \,\mathsf{R}\, \phi_2]\} & \text{if } \phi = \phi_1 \,\mathsf{R}\, \phi_2 \end{cases}$$

The computation of the cover of a set Φ comprises the inductive computation of the sets $elem_1$ and $elem_2$.

For example, suppose $\Phi = \{\mathsf{F}\psi\}$. Because $\mathsf{F}\psi$ is not a X formula, a proposition, or a negation of a proposition, it is non-elementary. Thus, we compute the sets $elem_1(\mathsf{F}\psi)$ and $elem_2(\mathsf{F}\psi)$. The former is $\{\psi\}$; the latter is $\{\mathsf{X}\mathsf{F}\psi\}$. Because the formulas in these two sets are elementary, there is no need for inductive computations of $elem_1$ and $elem_2$. The resulting cover looks as follows: $cover(\Phi) = \{\{\psi\}, \{\mathsf{X}\mathsf{F}\psi\}\}$. We now show that the required equivalence is true.

$$
\begin{aligned}
\textstyle\bigwedge_{\phi\in\Phi}\phi &\equiv \textstyle\bigwedge_{\phi\in\{\mathsf{F}\psi\}}\phi \\
&\equiv \mathsf{F}\psi \\
\\
\textstyle\bigvee_{C\in cover(\Phi)}\left[\bigwedge_{\phi'\in C}\phi'\right] &\equiv \textstyle\bigvee_{C\in\{\{\psi\},\{\mathsf{XF}\psi\}\}}\left[\bigwedge_{\phi'\in C}\phi'\right] \\
&\equiv \left[\textstyle\bigwedge_{\phi'\in\{\psi\}}\phi'\right] \vee \left[\bigwedge_{\phi'\in\{\mathsf{XF}\psi\}}\phi'\right] \\
&\equiv \psi \vee \mathsf{XF}\psi \\
&\equiv \mathsf{F}\psi
\end{aligned}
$$

Intuitively, each set in the cover of Φ represents one way of making the conjunction of these formulas true.

In practice, algorithms for computing covers implement some additional checks and optimisations for the sake of finding the *smallest* cover possible. See Sects. 2 and 4 of [29] for generic pseudo-code that can be instantiated for three different cover computation algorithms; in our implementation, we use the algorithm corresponding to Ltl2Aut. In fact, these additional checks and optimisations are the distinguishing factor of Ltl2Aut with respect to other algorithms.

Having defined and exemplified the notion of covers, we proceed with the definition of $\mathcal{A}_{\neg\varphi} = \langle \mathcal{Q}_{\neg\varphi}, \mathcal{I}_{\neg\varphi}, \delta_{\neg\varphi}, \mathcal{F}_{\neg\varphi}, \mathcal{D}_{\neg\varphi}, \Lambda_{\neg\varphi}\rangle$, which is derived from Fig. 2 and Sect. 3.2 in [29] (a formal definition of the automaton is not given there; only an algorithm and its informal treatise). Let $\neg\varphi$ be the negated property under investigation in what follows. We start with the definition of the automaton's structure, and proceed with the language it accepts.

A.2.1 Structure: $\mathcal{Q}_{\neg\varphi}, \mathcal{I}_{\neg\varphi}, \delta_{\neg\varphi}$

Each state $q_{\neg\varphi} \in \mathcal{Q}_{\neg\varphi}$ is a set of elementary formulas that occur in the cover of *the arguments of* X *formulas that occur in some other set of elementary formulas.* That is, the set of states $\mathcal{Q}_{\neg\varphi}$ is defined by inductive computation of the cover of the formulas that must be true *at the next time instant* for the formulas *in the current state* to be true, starting from the cover of $\{\neg\varphi\}$ (examples are given shortly). The transition function maps each state $q_{\neg\varphi} \in \mathcal{Q}_{\neg\varphi}$ to the set of states that correspond to the formulas that must be true at the next time instant. The automaton is augmented with a source state $\imath_{\neg\varphi}$ and a sink state $\jmath_{\neg\varphi}$ such that the structure of $\mathcal{A}_{\neg\varphi}$ is defined as follows.

$$
\begin{aligned}
\mathcal{Q}_{\neg\varphi} &= \{\imath_{\neg\varphi}, \jmath_{\neg\varphi}\} \cup cover(\{\neg\varphi\}) \cup \{C' \in 2^{\mathcal{L}_{\mathrm{LTL}}} \mid C \in \mathcal{Q}_{\neg\varphi} \\
&\qquad \text{and } next = \{\phi \in \mathcal{L}_{\mathrm{LTL}} \mid \mathsf{X}\phi \in C\} \\
&\qquad \text{and } C' \in cover(next)\} \\
\mathcal{I}_{\neg\varphi} &= \{\imath_{\neg\varphi}\} \\
\delta_{\neg\varphi}(q_{\neg\varphi}) &= \begin{cases}
cover(\{\neg\varphi\}) & \text{if } q_{\neg\varphi} = \imath_{\neg\varphi} \\
cover(\{\phi \in \mathcal{L}_{\mathrm{LTL}} \mid \mathsf{X}\phi \in q_{\neg\varphi}\}) & \text{if } q_{\neg\varphi} \in 2^{\mathcal{L}_{\mathrm{LTL}}} \\
& \text{and } cover(\{\phi \in \mathcal{L}_{\mathrm{LTL}} \mid \mathsf{X}\phi \in q_{\neg\varphi}\}) \neq \emptyset \\
\{\jmath_{\neg\varphi}\} & \text{otherwise}
\end{cases}
\end{aligned}
$$

The source state is there to ensure that $\mathcal{I}_{\neg\varphi}$ is a singleton (this will prove convenient when defining the product automaton). The reason for the sink state is to ensure that the automaton accepts infinite runs corresponding to computations with a *finite prefix* on which the truth of the LTL formula to which the automaton corresponds can be established. The addition of such a sink state is not explicitly mentioned in [29], but is apparent from Fig. 1 in [45].

To exemplify the previous, reconsider Fig. 3.1 that displays the Büchi automaton corresponding to $\mathsf{F}\psi$. To determine the set of states, we first compute the cover of $\{\mathsf{F}\psi\}$. Recall that $cover(\{\mathsf{F}\psi\}) = \{\{\psi\}, \{\mathsf{XF}\psi\}\}$, and let $q_0 = \{\mathsf{XF}\psi\}$ and $q_1 = \{\psi\}$. We proceed with inductive computation of covers for formulas that must be true one step forward in time. With respect to q_0, these formulas are $next = \{\mathsf{F}\psi\}$. With respect to q_1, we have $next = \emptyset$. Because we have already included the cover of $\{\mathsf{F}\psi\}$ as states (namely q_0 and q_1), neither q_0 nor q_1 introduce new states. The transition function maps the initial state $\imath$ to the cover of $\{\mathsf{F}\psi\}$, which are both q_0 and q_1.

The same holds for state q_0. In contrast, with respect to q_1, the set $cover(\{\,\phi \in \mathcal{L}_{\text{LTL}} \mid \mathsf{X}\phi \in q_1\,\})$ is empty (because $\{\,\phi \in \mathcal{L}_{\text{LTL}} \mid \mathsf{X}\phi \in q_1\,\} = \emptyset$) such that q_1 is mapped to $\jmath$. This is also the case for $\jmath$ itself.

A.2.2 Language: $\mathcal{F}_{\neg\varphi}, \mathcal{D}_{\neg\varphi}, \Lambda_{\neg\varphi}$

We proceed with the definition of the acceptance conditions $\mathcal{F}_{\neg\varphi}$, domain $\mathcal{D}_{\neg\varphi}$, and labelling function $\Lambda_{\neg\varphi}$. The acceptance conditions $F_{\neg\varphi} \in \mathcal{F}_{\neg\varphi}$ are chosen such that states in $F_{\neg\varphi}$ either do not "promise" to fulfil a U formula in the future, or fulfil it immediately. In this way, postponing forever fulfilling a promised U formula gives not rise to accepting runs [29]. In the definition below, let $untils(\phi)$ be the set of U (sub-)formulas in a formula ϕ, including all F (sub-)formulas (recall from Sect. 3.1.1 that $\mathsf{F}\phi = \top\,\mathsf{U}\,\phi$). The domain $\mathcal{D}_{\neg\varphi}$ is the power set of all the atomic propositions, i.e. $2^{\mathcal{P}}$ in general, i.e. $2^{\mathcal{L}_\Psi}$ for GOAL. Note that the domain of the property automaton is the same as the domain of the program automaton. Finally, the labelling function associates each state $q_{\neg\varphi}$ with the subsets of $\mathcal{D}_{\neg\varphi}$ that are *compatible* with the propositional information contained in $q_{\neg\varphi}$ [29]. Note that if $\bot$ is in $q_{\neg\varphi}$, then no subset can every be compatible with it. We now have the following.

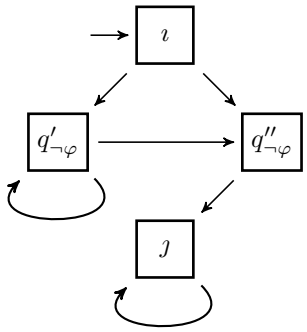
$$
\begin{aligned}
\mathcal{F}_{\neg\varphi} &= \textstyle\bigcup_{\phi_1\mathsf{U}\phi_2 \in untils(\neg\varphi)} \{\,\{\,q_{\neg\varphi} \in \mathcal{Q}_{\neg\varphi} \mid \phi_2 \in q_{\neg\varphi} \text{ or } \mathsf{X}[\phi_1\,\mathsf{U}\,\phi_2] \notin q_{\neg\varphi}\,\}\,\} \\
\mathcal{D}_{\neg\varphi} &= 2^{\mathcal{L}_\Psi} \\
\Lambda_{\neg\varphi}(q_{\neg\varphi}) &= \begin{cases}
\{\,\Psi \in \mathcal{D}_{\neg\varphi} \mid \mathit{props} = q_{\neg\varphi} \cap \mathcal{L}_\Psi \text{ and } \mathit{props} \subseteq \Psi \text{ and } \neg\psi \notin q_{\neg\varphi} \text{ for all } \psi \in \Psi\,\} & \text{if } q_{\neg\varphi} \in 2^{\mathcal{L}_{\text{LTL}}} \text{ and } \bot \notin q_{\neg\varphi} \\
\emptyset & \text{if } q_{\neg\varphi} \in 2^{\mathcal{L}_{\text{LTL}}} \text{ and } \bot \in q_{\neg\varphi} \\
\mathcal{D}_{\neg\varphi} & \text{if } q_{\neg\varphi} \in \{\imath_{\neg\varphi}, \jmath_{\neg\varphi}\}
\end{cases}
\end{aligned}
$$

As an example of these definitions, reconsider the automaton for $\mathsf{F}\psi$ given in Fig. 3.1. First, note that $untils(\mathsf{F}\psi) = \{\top\,\mathsf{U}\,\psi\}$, which is why only a single acceptance condition is imposed. We now consider all states of the automaton with respect to this acceptance condition. First, both $\imath$ and $\jmath$ are trivially in it, because $\imath$ and $\jmath$ are *dummies* rather than sets of mental states. State $q_0 = \{\mathsf{XF}\psi\}$ "promisses" to fulfill $\mathsf{XF}\psi = \top\,\mathsf{U}\,\psi$ in the next state such that it violates the second requirement, and neither does it satisfy the first requirement. Hence, q_0 is not in the acceptance condition. Finally, state $q_1 = \{\psi\}$ immediately fulfils $\top\,\mathsf{U}\,\psi$ such that it satisfies the first requirement, and hence is in the acceptance condition. The domain is the power set of the language of mental state conditions such that it at least contains sets of which ψ is a member. Finally, the labelling function labels the source state and the sink state with the entire domain. The same holds for q_0, because its propositional part is empty. In contrast, q_1 is labelled with a subset of the labels in the domain, namely those that contain at least ψ (this is because $\mathit{props} = \{\psi\}$ for q_1).

A.2.3 Example: `blenderAgent`

We discuss a second example. Suppose we want to verify whether `blenderAgent` eventually believes to have ticked the "bananas" check box on the recipe, i.e. $\varphi = \mathsf{F}\,\mathbf{bel}(\texttt{ticked(bananas)})$. Recall from Sect. 3.2 that if we want to verify whether an agent P satisfies a property φ, we check the intersection of the automaton $\mathcal{A}_P$ with the automaton $\mathcal{A}_{\neg\varphi}$ corresponding to the *negated* property $\neg\varphi$. Thus, in this example, we construct an automaton for $\neg\varphi = \mathsf{G}\neg\mathbf{bel}(\texttt{ticked(bananas)})$, and denote this automaton by $\mathcal{A}_{\neg\varphi} = \langle \mathcal{Q}_{\neg\varphi}, \mathcal{I}_{\neg\varphi}, \delta_{\neg\varphi}, \mathcal{F}_{\neg\varphi}, \mathcal{D}_{\neg\varphi}, \Lambda_{\neg\varphi}\rangle$. Note that $\neg\varphi$ is identical to the property that we assumed in the example of the program automaton corresponding to `blenderAgent` in Appx. A.1.

We start with $\mathcal{A}_{\neg\varphi}$'s structure, i.e. the set of states $\mathcal{Q}_{\neg\varphi}$, initial states $\mathcal{I}_{\neg\varphi}$, and transition function $\delta_{\neg\varphi}$. The cover of $\{\,\mathsf{G}\neg\mathbf{bel}(\texttt{ticked(bananas)})\,\}$ is obtained after a single decomposition step, i.e. the set $\{\{\bot\}, \{\neg\mathbf{bel}(\texttt{ticked(bananas)}), \mathsf{XG}\neg\mathbf{bel}(\texttt{ticked(bananas)})\}\}$.



$$
\begin{aligned}
\mathcal{Q}_{\neg\varphi} &= \{\imath, q'_{\neg\varphi}, q''_{\neg\varphi}, \jmath\} \\
\mathcal{I}_{\neg\varphi} &= \{\imath\} \\
\delta_{\neg\varphi}(q_{\neg\varphi}) &= \begin{cases} \{q'_{\neg\varphi}, q''_{\neg\varphi}\} & \text{if } q_{\neg\varphi} \in \{\imath, q'_{\neg\varphi}\} \\ \{\jmath\} & \text{if } q_{\neg\varphi} \in \{q''_{\neg\varphi}, \jmath\} \end{cases} \\
\mathcal{F}_{\neg\varphi} &= \emptyset \\
\mathcal{D}_{\neg\varphi} &= \{\emptyset, \{\mathbf{bel}(\texttt{ticked(bananas)})\}\} \\
\Lambda_{\neg\varphi}(q_{\neg\varphi}) &= \begin{cases} \mathcal{D}_{\neg\varphi} & \text{if } q_{\neg\varphi} \in \{\imath, \jmath\} \\ \{\emptyset\} & \text{if } q_{\neg\varphi} = q'_{\neg\varphi} \\ \emptyset & \text{if } q_{\neg\varphi} = q''_{\neg\varphi} \end{cases}
\end{aligned}
$$

Figure A.2: Büchi automaton for $\mathsf{G}\neg\mathbf{bel}(\texttt{ticked(bananas)})$.

From this, it follows that there are 2 states (in addition to the source state and sink state): $q'_{\neg\varphi} = \{\neg\mathbf{bel}(\texttt{ticked(bananas)}), \mathsf{X}\mathsf{G}\neg\mathbf{bel}(\texttt{ticked(bananas)})\}$ and $q''_{\neg\varphi} = \{\bot\}$. As always, the initial states $\mathcal{I}_{\neg\varphi}$ comprise the source state only, which is mapped to $\{q'_{\neg\varphi}, q''_{\neg\varphi}\}$ by the transition function. The same is true for $q'_{\neg\varphi}$, i.e. $\delta_{\neg\varphi}(q'_{\neg\varphi}) = \{q'_{\neg\varphi}, q''_{\neg\varphi}\}$. Finally, because the cover of $\bot$ is empty, $\delta_{\neg\varphi}(q''_{\neg\varphi}) = \{\jmath_{\neg\varphi}\}$. The same is, by definition, true for $\jmath_{\neg\varphi}$. Figure A.2 displays $\mathcal{A}_{\neg\varphi}$ graphically. Note that the structure is equal to the structure of the automaton corresponding to $\mathsf{F}\psi$ in Fig. 3.1. The acceptance conditions, domain, and labelling function are, in contrast, different.

That is, with respect to the acceptance conditions $\mathcal{F}_{\neg\varphi}$, note that no U formulas occur in $\neg\varphi$. Hence, $\mathcal{F}_{\neg\varphi} = \emptyset$ meaning that *any* infinite run of this automaton is accepting. The domain $\mathcal{D}_{\neg\varphi}$ is shown in Fig. A.2 to contain the empty set $\emptyset$ and a singleton containing $\mathbf{bel}(\texttt{ticked(bananas)})$. What happened to the power set of $\mathcal{L}_\Psi$? Well, as remarked earlier, due to the infiniteness of $\mathcal{L}_\Psi$, it is impossible to use its entire power set as domain in practice. Instead, we use a subset of $2^{\mathcal{L}_\Psi}$ that is *sufficiently large*. By Prop. 1.2 in [76], the power set of the set containing all atomic propositions that occur in $\neg\varphi$ is sufficiently large. Let $voc(\phi)$ be the set of all atomic propositions that occur in an LTL formula ϕ, and call this set the *vocabulary* of ϕ [76]. Then, the domain is $2^{voc(\neg\varphi)}$. In our current example (as well as in the example about the program automaton corresponding to $\texttt{blenderAgent}$), the vocabulary is $\{\mathbf{bel}(\texttt{ticked(bananas)})\}$, such that the domain indeed is $\{\emptyset, \{\mathbf{bel}(\texttt{ticked(bananas)})\}\}$. Note that the use of the vocabulary is "merely" an implementation detail (and, in written examples like our current, notationally convenient); the theory itself is not changed. Given the domain, the labelling function $\Lambda_{\neg\varphi}$ is as follows. With respect to the source and sink state, by definition, the labelling consists of the entire domain. State $q'_{\neg\varphi}$ is labelled with the empty set, because the label $\{\mathbf{bel}(\texttt{ticked(bananas)})\}$ contains a formula that occurs negated in $q'_{\neg\varphi}$ (recall $q'_{\neg\varphi} = \{\neg\mathbf{bel}(\texttt{ticked(bananas)}), \mathsf{X}\mathsf{G}\neg\mathbf{bel}(\texttt{ticked(bananas)})\}$). Finally, $q''_{\neg\varphi}$ is not assigned a label at all (notice the difference between $\{\emptyset\}$ and $\emptyset$). The reason is that $\bot$ occurs in $q''_{\neg\varphi}$ which is incompatible with any proposition.

A.3 Product automata

Having defined the program and property automata, we proceed with the *product automaton*, denoted by $\mathcal{A}_\times$ and defined as the tuple $\mathcal{A}_\times = \langle \mathcal{Q}_\times, \mathcal{I}_\times, \delta_\times, \mathcal{F}_\times, \mathcal{D}_\times, \Lambda_\times \rangle$. Recall from Sect. 3.2 that we want $\mathcal{A}_\times$ to satisfy the following property: $L(\mathcal{A}_\times) = L(\mathcal{A}_P) \cap L(\mathcal{A}_{\neg\varphi})$. Such an automaton $\mathcal{A}_\times$ actually exists, because Büchi automata are closed under intersection [17]. We first give the formal definition of $\mathcal{A}_\times$ (which already appeared without explanation in Sect. 3.2.1), given a program automaton $\mathcal{A}_P = \langle \mathcal{Q}_P, \mathcal{I}_P, \delta_P, \mathcal{F}_P, \mathcal{D}_P, \Lambda_P \rangle$, and a property automaton $\mathcal{A}_{\neg\varphi} = \langle \mathcal{Q}_{\neg\varphi}, \mathcal{I}_{\neg\varphi}, \delta_{\neg\varphi}, \mathcal{F}_{\neg\varphi}, \mathcal{D}_{\neg\varphi}, \Lambda_{\neg\varphi} \rangle$. After the definition, derived from [92], we explain its meaning.

$$
\begin{array}{lcl}
\mathcal{Q}_\times & = & \mathcal{Q}_P \times \mathcal{Q}_{\neg\varphi} \\
\mathcal{I}_\times & = & \mathcal{I}_P \times \mathcal{I}_{\neg\varphi} \\
\\
\delta_\times(q_\times) & = & \begin{cases} \delta_P(q_P) \times \delta_{\neg\varphi}(q_{\neg\varphi}) & \text{if } q_\times = \langle q_P, q_{\neg\varphi}\rangle \\ & \text{and } \Lambda_P(q_P) \cap \Lambda_{\neg\varphi}(q_{\neg\varphi}) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\
\\
\mathcal{F}_\times & = & \widehat{\mathcal{F}_P} \cup \widehat{\mathcal{F}_{\neg\varphi}} \text{ in which } \widehat{\mathcal{F}_P} = \bigcup_{F_P \in \mathcal{F}_P} \{ F_P \times \mathcal{Q}_{\neg\varphi} \} \\
& & \text{and } \widehat{\mathcal{F}_{\neg\varphi}} = \bigcup_{F_{\neg\varphi} \in \mathcal{F}_{\neg\varphi}} \{ \mathcal{Q}_P \times F_{\neg\varphi} \} \\
\\
\mathcal{D}_\times & = & 2^{\mathcal{L}_\Psi} \\
\Lambda_\times(q_\times) & = & \Lambda_P(q_P) \cap \Lambda_{\neg\varphi}(q_{\neg\varphi}) \text{ in which } q_\times = \langle q_P, q_{\neg\varphi}\rangle
\end{array}
$$

$\mathcal{Q}_\times$ is defined as the product of the states in the individual automata, i.e. a state $q_\times$ is a pair $q_\times = \langle q_P, q_{\neg\varphi}\rangle$. $\mathcal{I}_\times$ is the singleton $\mathcal{I}_\times = \{\langle \imath_P, \imath_{\neg\varphi}\rangle\}$, because both the program and property automata have a source state as the only initial state. This is convenient because the algorithm that checks for emptiness of the language of a Büchi automaton assumes a single initial state (see Appx. A.3). The transition function $\delta_\times$ is called *synchronised*, because transitions are *joint* [58]: they only occur if the sets of labels on the states q_P and $q_{\neg\varphi}$ have a common element. In [92] (from which our definition is derived), labels are on transitions rather than on states, which is why synchronisation is expressed slightly differently in [92]'s definition. The intuition behind this synchronisation is, however, the same: the property automaton *constrains* the computations that the program automaton represents. That is, every computation of an agent P is represented by a run of the corresponding Büchi automaton $\mathcal{A}_P$, and each state in $\mathcal{A}_P$ is labelled with the set of mental state conditions that are true in it. The states of the property automaton $\mathcal{A}_{\neg\varphi}$ corresponding to the negated property $\neg\varphi$ are labelled with all the sets of mental state conditions that may be true in that state for $\neg\varphi$ to be true. Thus, if the label on a state q_P of $\mathcal{A}_P$ is *not* a label of a state $q_{\neg\varphi}$ in $\mathcal{A}_{\neg\varphi}$, then the computations that go through q_P cannot satisfy $\neg\varphi$, and $\mathcal{A}_\times$ hence should not accept runs on which $\langle q_P, q_{\neg\varphi}\rangle$ occurs. This is achieved by not giving $\langle q_P, q_{\neg\varphi}\rangle$ successors (i.e. a state without successors can never occur on an infinite run). The definition of the set of acceptance conditions $\mathcal{F}_\times$ is straightforward: every acceptance condition of the one automaton (F_P or $F_{\neg\varphi}$) is multiplied with all the states in the other ($\mathcal{Q}_{\neg\varphi}$ or $\mathcal{Q}_P$). The domain is the same as the domain of the two individual automata (whose domains are equal), and the labelling function makes sure that the language of $\mathcal{A}_\times$ is indeed the intersection of the languages of $\mathcal{A}_P$ and $\mathcal{A}_{\neg\varphi}$.

We illustrate the definition with an example: the product of the program automaton corresponding to `blenderAgent` (see Fig. A.1), denoted $\mathcal{A}_P$ in the remainder, and the property automaton corresponding to `G¬bel(ticked(bananas))` (see Fig. A.2), denoted $\mathcal{A}_{\neg\varphi}$ in the remainder. Their product automaton, denoted $\mathcal{A}_\times$ in the remainder, appears in Fig. A.3. The figure does not show states in $\mathcal{Q}_\times$ that are not reachable from the initial state $\langle \imath_P, \imath_{\neg\varphi}\rangle$. As the definitions of $\mathcal{Q}_\times$ and $\mathcal{I}_\times$ are straightforward, we focus on the definition of the transition function.

Let us start with the initial state $\langle \imath_P, \imath_{\neg\varphi}\rangle$. Because $\Lambda_P(\imath_P) = \Lambda_{\neg\varphi}(\imath_{\neg\varphi})$, it is true that $\Lambda_P(\imath_P) \cap \Lambda_{\neg\varphi}(\imath_{\neg\varphi}) \neq \emptyset$ such that the transition function for this product state is $\delta(\langle \imath_P, \imath_{\neg\varphi}\rangle) = \delta(\imath_P) \times \delta(\imath_{\neg\varphi}) = \{\mu_0\} \times \{q'_{\neg\varphi}, q''_{\neg\varphi}\} = \{\langle \mu_0, q'_{\neg\varphi}\rangle, \langle \mu_0, q''_{\neg\varphi}\rangle\}$. Let us first consider the latter of these two states, i.e. $\langle \mu_0, q''_{\neg\varphi}\rangle$. Because the labelling function of the property automaton $\Lambda_{\neg\varphi}$ maps $q''_{\neg\varphi}$ to the empty set, *any* intersection with $\Lambda_{\neg\varphi}(q''_{\neg\varphi})$ is empty. Consequently, $\delta_\times(\langle \mu_0, q''_{\neg\varphi}\rangle) = \emptyset$. With respect to the former state, i.e. $\langle \mu_0, q'_{\neg\varphi}\rangle$, both $\Lambda_P(\mu_0)$ and $\Lambda_{\neg\varphi}(q'_{\neg\varphi})$ are $\{\emptyset\}$ such that $\Lambda_P(\mu_0) \cap \Lambda_{\neg\varphi}(q'_{\neg\varphi}) \neq \emptyset$. Hence, $\Lambda_\times(\langle \mu_0, q'_{\neg\varphi}\rangle) = \delta(\mu_0) \times \delta(q'_{\neg\varphi}) = \{\mu_1\} \times \{q'_{\neg\varphi}, q''_{\neg\varphi}\} = \{\langle \mu_1, q'_{\neg\varphi}\rangle, \langle \mu_1, q''_{\neg\varphi}\rangle\}$. Similar reasoning can be applied to the definition of the transition function in all other states.

Because $\mathcal{F}_P = \{\mathcal{Q}_P\}$ (see Fig. A.1) and $\mathcal{F}_{\neg\varphi} = \emptyset$ (see Fig. A.2), the set of acceptance conditions of the product automaton $\mathcal{F}_\times = \{\mathcal{Q}_P \times \mathcal{Q}_{\neg\varphi}\} \cup \{\mathcal{Q}_P \times \emptyset\} = \{\mathcal{Q}_\times\} \cup \emptyset = \{\mathcal{Q}_\times\}$. Thus, every state is an accepting state of the only acceptance condition such that every infinite run is accepting. However, $\mathcal{A}_\times$ has no infinite runs, because its state space is *acyclic* due to the definition

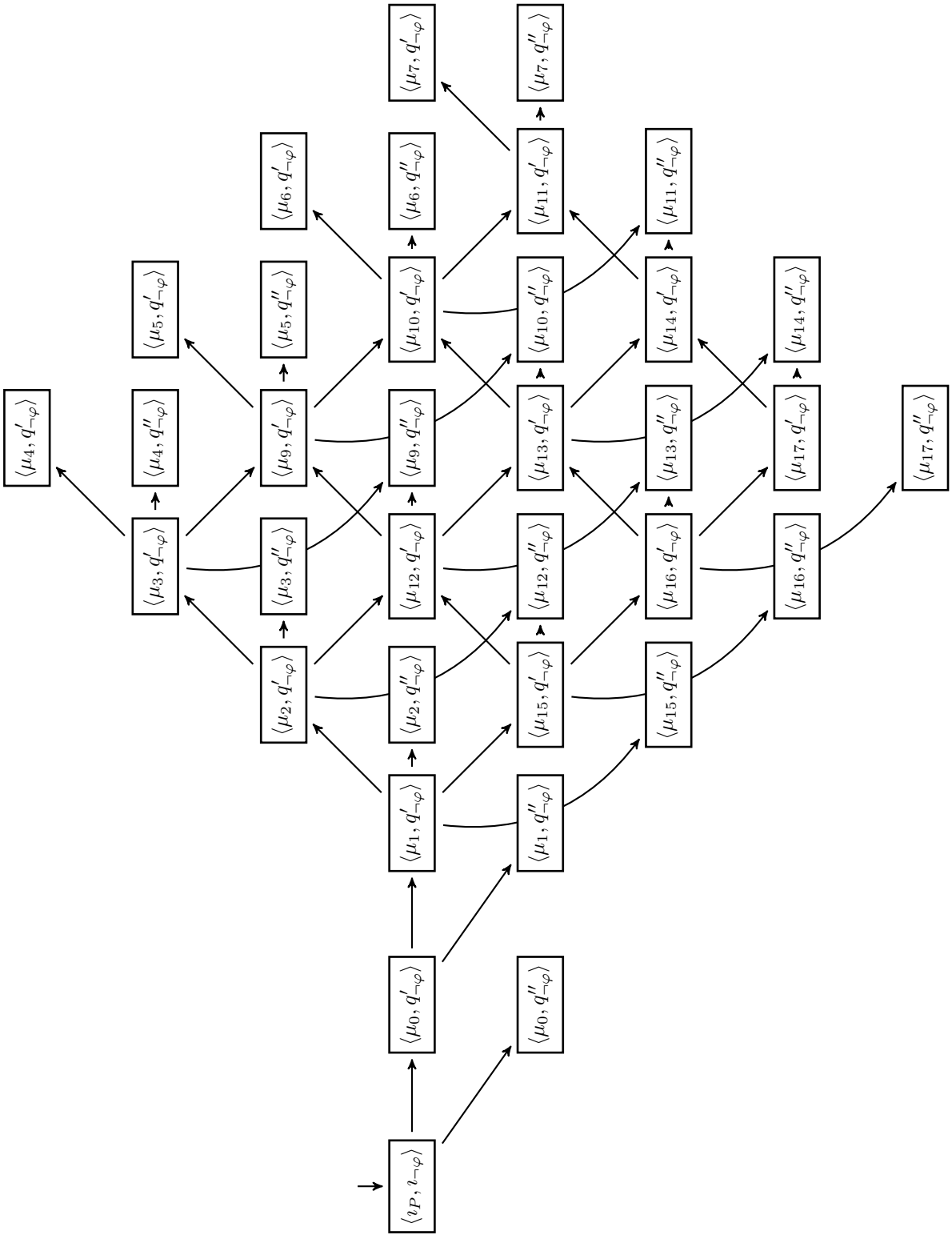


Figure A.3: Büchi automaton for the product of Figs. A.1 and A.2.

of the transition function (this is well observable in Fig. A.3). If the automaton contains no cycles, it only has finite runs, hence an accepting state cannot occur infinitely often on such a run. No accepting runs implies no accepted words such that the language of $\mathcal{A}_\times$ is empty. This means that no computation of `blenderAgent` is a model of $\neg\varphi = \mathsf{G}\neg\mathbf{bel}(\texttt{ticked(bananas)})$. Hence, all its computations are models of $\varphi = \mathsf{F}\,\mathbf{bel}(\texttt{ticked(bananas)})$, i.e. `blenderAgent` believes to have ticked the "bananas" check box on the recipe eventually.

A.4 Language Emptiness

Finally, we describe [90, 91]'s NDFS algorithm for determining language emptiness of a Büchi automaton $\mathcal{A} = \langle \mathcal{Q}, \mathcal{I}, \delta, \mathcal{F}, \mathcal{D}, \Lambda \rangle$. We refer to Sect. 3.2.2 for our motivation to choose this particular algorithm. The algorithm is given as Algs. 7 and 8, and uses the following data structures:

- *conds* is a mapping $\mathcal{Q} \longrightarrow 2^{\mathcal{F}}$ that maps states to an initially empty subset of acceptance conditions. Informally, *conds* expresses that if $F \in \mathit{conds}[q]$ for some $F \in \mathcal{F}$ and $q \in \mathcal{Q}$, then q is reachable in the automaton from a state q', for which holds $q' \in F$.
- *path* is a stack of states $q \in \mathcal{Q}$ that represent the current path in the automaton, starting at the initial state $\imath$ (when applied to a product automaton as we do in this thesis, $\imath = \langle \imath_P, \imath_{\neg\varphi} \rangle$).
- *processed* is a set of states that have already been processed.

The algorithm works as follows. On lines 1-4, the search is initialised: the algorithm initially knows nothing about the acceptance conditions and reachability of states, and the path starts at $\imath$. The depth-first traversal is implemented by lines 7-10: as long as some current state q has an unvisited successor q', this successor is added to *path*, and the search continues from q'. Due to

---

1: *conds* := a mapping $\mathcal{Q} \longrightarrow 2^{\mathcal{F}}$ such that initially, $\mathit{conds}[q] = \emptyset$ for all $q \in \mathcal{Q}$
2: *path* := an empty stack
3: push $\imath$ on *path*
4: $\mathit{processed} = \emptyset$

5: **while** *path* is not empty **do**
6: q := the top state on *path*
7: **while** there exists a $q' \in \delta(q)$ such that $q' \notin \mathit{path} \cup \mathit{processed}$ **do**
8: push q' on the *path* stack
9: $q := q'$
10: **end while**
11: **if** $\mathit{conds}[q] \neq \emptyset$ **or** there exists a $F \in \mathcal{F}$ such that $q \in F$ **then**
12: apply Alg. 8 to $\mathcal{A}$, $\{q\}$, and $\mathit{conds}[q] \cup \{F \in \mathcal{F} \mid q \in F\}$
13: **if** $\mathit{conds}[q] = \mathcal{F}$ **then**
14: **return false**
15: **end if**
16: **end if**
17: $\mathit{processed} := \mathit{processed} \cup \{q\}$
18: pop the top state off *path*
19: **end while**

20: **return true**

Algorithm 7: Let $\mathcal{A} = \langle \mathcal{Q}, \mathcal{I}, \delta, \mathcal{F}, \mathcal{D}, \Lambda \rangle$ be a Büchi automaton such that $\mathcal{I}$ is a singleton, i.e. $\mathcal{I} = \{\imath\}$. Then, this algorithm return **true** if the language $L(\mathcal{A})$ of this automaton is empty, and **false** otherwise.

the way the traversal is implemented, it is crucial that the top of *path* is only *peeked* rather than popped on line 6; if it would be popped, then the algorithm would backtrack all the way to $\imath$ after the exploration of a single path. Lines 11-16 define what happens when the search detects a cycle, i.e. a state that is already on the current search stack is re-encountered. If:

- either the automaton contains a cycle from q to itself, such that the cycle fulfils at least one acceptance condition F (see Lemma 2 of [91] for a detailed proof),
- or q is part of at least one acceptance condition itself,

...then a second search is initiated by means of an application of Alg. 8. The task of this second search is to update the *conds*-mapping of the states that are reachable from the states in *states*. That is, if a state q' is reachable from q, then $conds[q']$ must be updated such that after the second search it at least contains the elements of $conds[q]$ (but possibly more, namely the conditions that were already in $conds[q']$ before the second search). To avoid redundant work, the second search will not explore states q' for which $toPropagate \in 2^{\mathcal{F}}$ is already a subset of $conds[q']$.

If, after termination of Alg. 8, $conds[q]$ equals $\mathcal{F}$, then the algorithm terminates, because the language of the automaton is non-empty (lines 13-15). To see this, observe that all F that were in $conds[q]$ before the second search, were in fact already fulfilled before that search (by Lemma 2 of [91]). The remaining acceptance conditions are those that contain q itself. If $conds[q] = \mathcal{F}$ after the second search, then that search must have traversed a cycle from q to itself, since $conds[q]$ cannot have been updated otherwise ($conds[q]$ is not updated before or at the beginning of the second search). Therefore, if $conds[q] = \mathcal{F}$ after the second search, then q fulfils all acceptance conditions in $\mathcal{F}$ (that is, q is on a cycle that goes through at least one state of each $F \in \mathcal{F}$). For a complete proof of correctness, consult [91].

Comparing Algs. 7 and 8 to the algorithms given in Fig. 2 of [91] yields the following differences. First, the "reset" statement on line 8 of Fig. 2 is not included in Alg. 7. This is justified, because this statement is only necessary in case of imperfect hashing of states, which we will not do (see the proof of Lemma 2 of [91]). For similar reasons, we omitted lines 13-15 and 29-30; see the paragraph above Lemma 3 in [91]. In that same paragraph, it is also remarked that it is allowed to replace line 17 of Fig. 2 by our line 17.

```
21: while states ≠ ∅ do
22:     removes a state q from states
23:     while there exists a q′ ∈ path∪processed such that q′ ∈ δ and toPropagate ⊈ conds[q′]
        do
24:         states := states ∪ {q′}
25:         conds[q′] := conds[q′] ∪ toPropagate
26:     end while
27: end while
```

Algorithm 8: Let $\mathcal{A} = \langle \mathcal{Q}, \mathcal{I}, \delta, \mathcal{F}, \mathcal{D}, \Lambda \rangle$ be a Büchi automaton, let $states \subseteq \mathcal{Q}$ be a set of states, and let $toPropagate \subseteq \mathcal{F}$ be a set of acceptance conditions. Additionally, let *conds*, *path*, and *processed* be globally accessible from Alg. 7. Then: this algorithms extends the *conds* mapping for the states in *states* and their already processed successors with the conditions in $toPropagate$.

Appendix B

Experimental Results

This appendix provides the raw experimental results that were treated and analysed in Chap. 4. In the tables that follow, verification / execution times are given in milliseconds (ms), seconds (s), minutes (min), or hours (h); memory consumption is given in megabytes (MB). In the first four tables, i.e. Tables B.1,B.2,B.3,B.4, resource consumption during model checking is given. The second series of tables, i.e. Tables B.5,B.6,B.7,B.8 provide resource consumption during execution.

n	Verification time			Memory consumption		
	IMC	AJPF	MMC	IMC	AJPF	MMC
10	222 ms	13 s	8 s	4 MB	53 MB	14.9 MB
20	295 ms	25 s	53 s	7 MB	70 MB	15.1 MB
30	353 ms	46 s	3 min	7 MB	95 MB	15.7 MB
45	401 ms	79 s	12 min	7 MB	112 MB	17.0 MB
60	560 ms	157 s	32 min	13 MB	105 MB	16.8 MB
80	496 ms	5 min	88 min	13 MB	132 MB	18.2 MB
100	852 ms	7 min	3 h	13 MB	131 MB	19.8 MB
200	1123 ms	37 min	44 h	13 MB	182 MB	29.4 MB

Table B.1: Resource consumption during verification – Experiment 1 (Blocks World)

n	Verification time			Memory consumption		
	IMC	AJPF	MMC	IMC	AJPF	MMC
10	131 ms	9 s	831 ms	4 MB	43 MB	14.7 MB
20	186 ms	15 s	4 s	7 MB	52 MB	15.0 MB
30	248 ms	22 s	12 s	7 MB	70 MB	15.7 MB
45	261 ms	40 s	34 s	7 MB	81 MB	17.0 MB
60	484 ms	69 s	75 s	7 MB	104 MB	16.2 MB
80	403 ms	158 s	164 s	13 MB	82 MB	17.7 MB
100	509 ms	4 min	5 min	13 MB	82 MB	19.3 MB
200	968 ms	20 min	35 min	13 MB	77 MB	28.0 MB

Table B.2: Resource consumption during verification – Experiment 2 (Buggy Blocks World)

n	Verification time			Memory consumption		
	IMC	AJPF	MMC	IMC	AJPF	MMC
10	228 ms	10 s	302 ms	4 MB	51 MB	15.1 MB
20	385 ms	12 s	611 ms	7 MB	60 MB	15.6 MB
30	516 ms	14 s	867 ms	13 MB	60 MB	15.9 MB
45	761 ms	18 s	1319 ms	13 MB	73 MB	16.2 MB
60	948 ms	22 s	1849 ms	13 MB	87 MB	16.7 MB
80	1195 ms	27 s	2381 ms	15 MB	94 MB	17.2 MB
100	1460 ms	33 s	3 s	25 MB	102 MB	17.8 MB
200	2635 ms	72 s	7 s	26 MB	187 MB	20.6 MB

Table B.3: Resource consumption during verification – Experiment 3 (Counting)

n	Verification time			Memory consumption		
	IMC	AJPF	MMC	IMC	AJPF	MMC
10	314 ms	10 s	1212 ms	4 MB	50 MB	15.2 MB
20	430 ms	14 s	5 s	7 MB	51 MB	15.5 MB
30	747 ms	17 s	15 s	12 MB	58 MB	15.7 MB
45	1096 ms	23 s	44 s	12 MB	78 MB	16.1 MB
60	1089 ms	31 s	99 s	12 MB	90 MB	16.5 MB
80	1653 ms	46 s	4 min	21 MB	108 MB	18.1 MB
100	1924 ms	63 s	7 min	24 MB	132 MB	19.6 MB
200	2589 ms	5 min	69 min	24 MB	226 MB	30.7 MB

Table B.4: Resource consumption during verification – Experiment 4 (Counting and Memorizing)

n	Execution time			Memory consumption		
	IMC	AJPF	MMC	IMC	AJPF	MMC
10	103 ms	103 ms	3 s	3 MB	3 MB	14.8 MB
20	171 ms	298 ms	17 s	3 MB	3 MB	15.0 MB
30	225 ms	445 ms	47 s	5 MB	3 MB	15.2 MB
45	377 ms	569 ms	136 s	6 MB	4 MB	15.6 MB
60	363 ms	728 ms	5 min	4 MB	7 MB	16.1 MB
80	468 ms	886 ms	11 min	5 MB	10 MB	17.5 MB
100	694 ms	1000 ms	20 min	9 MB	16 MB	19.0 MB
200	1025 ms	1395 ms	145 min	12 MB	57 MB	27.3 MB

Table B.5: Resource consumption during execution – Experiment 1 (Blocks World)

n	Execution time			Memory consumption		
	IMC	AJPF	MMC	IMC	AJPF	MMC
10	75 ms	66 ms	3 s	2 MB	3 MB	14.8 MB
20	124 ms	161 ms	14 s	3 MB	3 MB	15.1 MB
30	185 ms	417 ms	38 s	2 MB	3 MB	15.7 MB
45	211 ms	467 ms	108 s	3 MB	3 MB	16.9 MB
60	290 ms	563 ms	4 min	4 MB	3 MB	16.2 MB
80	344 ms	601 ms	9 min	4 MB	10 MB	17.5 MB
100	424 ms	760 ms	16 min	9 MB	15 MB	19.1 MB
200	742 ms	1221 ms	120 min	12 MB	57 MB	27.4 MB

Table B.6: Resource consumption during execution – Experiment 2 (Buggy Blocks World)

n	Execution time			Memory consumption		
	IMC	AJPF	MMC	IMC	AJPF	MMC
10	185 ms	86 ms	264 ms	3 MB	3 MB	15 MB
20	320 ms	132 ms	485 ms	6 MB	3 MB	15 MB
30	432 ms	173 ms	752 ms	7 MB	4 MB	15 MB
45	643 ms	273 ms	1072 ms	9 MB	4 MB	15 MB
60	752 ms	317 ms	1690 ms	12 MB	4 MB	15 MB
80	960 ms	381 ms	2073 ms	13 MB	4 MB	15 MB
100	1209 ms	469 ms	2389 ms	20 MB	5 MB	15 MB
200	2138 ms	865 ms	5 s	25 MB	15 MB	15 MB

Table B.7: Resource consumption during execution – Experiment 3 (Counting)

n	Execution time			Memory consumption		
	IMC	AJPF	MMC	IMC	AJPF	MMC
10	190 ms	111 ms	891 ms	3 MB	3 MB	15 MB
20	325 ms	203 ms	4 s	6 MB	3 MB	15 MB
30	510 ms	328 ms	12 s	7 MB	4 MB	15 MB
45	626 ms	425 ms	36 s	9 MB	4 MB	15 MB
60	825 ms	495 ms	81 s	12 MB	4 MB	15 MB
80	1122 ms	641 ms	3 min	13 MB	4 MB	15 MB
100	1413 ms	728 ms	6 min	20 MB	7 MB	15 MB
200	2223 ms	1495 ms	45 min	25 MB	23 MB	15 MB

Table B.8: Resource consumption during execution – Experiment 4 (Counting and Memorizing)

Appendix C

Proofs

C.1 Approximate Write Sets

In this section, we prove a series of propositions and lemmas on which the proofs of Theorems 1,2,3 in Sect. 5.3.2 are built. These theorems state that the definitions given in Sect. 5.3 yield proper over-approximations of write sets. That is, approximate write sets with respect to a set of mental state conditions Ψ for a transition class $\boldsymbol{\tau}(\rho)$ satisfy the following for all $\tau \in \boldsymbol{\tau}(\rho)$:

$$\begin{array}{lcl} \mathsf{Write}^{+}(\tau, \Psi) & \subseteq & \mathsf{ApproxWrite}^{+}(\boldsymbol{\tau}(\rho), \Psi) \\ \mathsf{Write}^{-}(\tau, \Psi) & \subseteq & \mathsf{ApproxWrite}^{-}(\boldsymbol{\tau}(\rho), \Psi) \\ \mathsf{Write}(\tau, \Psi) & \subseteq & \mathsf{ApproxWrite}(\boldsymbol{\tau}(\rho), \Psi) \end{array}$$

The exposition in this section follows the same structure that is also present in the definition of approximate write sets. That is, in the definition of $\mathsf{Write}^{+}(\tau, \Psi)$ and $\mathsf{Write}^{-}(\tau, \Psi)$ in Def. 20, the *Der* relations occur. For proving Theorems 1,2,3, we prove two lemmas about Der^{+} and Der^{-} in Appx. C.1.3 that in Sect. 5.3.2 are used to prove these theorems. Likewise, in the definitions of Der^{+} and Der^{-} in Defs. 18,19, the *der* relations occur. The lemmas that we prove about der^{+} and der^{-} in Appx. C.1.2 are used to prove the lemmas about the *Der* relations. Finally, because der^{+} and der^{-} are applied to the result of the approximate add/del function $\mathsf{Approx}\Delta$ defined in Defs. 11,12,13,15, we start with proving a series of propositions about this function's result when applied to a transition class in Appx. C.1.1. The dependencies between propositions, lemmas, and theorems discussed in this section are shown schematically in Fig. C.1.

C.1.1 ApproxΔ Propositions

In this subsection, we present four propositions: the first two propositions concern the sets $\mathsf{ApproxAdd}_{\Sigma}$ and $\mathsf{ApproxDel}_{\Sigma}$, and the last two propositions concern the sets $\mathsf{ApproxAdd}_{\Gamma}$ and $\mathsf{ApproxDel}_{\Gamma}$. Informally, these propositions express the following. Let $\boldsymbol{\tau}(\rho)$ be a transition class, and let $\tau \in \boldsymbol{\tau}(\rho)$. Then, all the facts that become derivable or underivable from the belief and goal bases after execution of τ are contained in the result of $\mathsf{Approx}\Delta$.

We remind the reader that the entailment relation $\models$ denotes Prolog's entailment relation. Let X be a set of clauses and let χ be a query. Then, informally, $X \models \chi$ is true iff evaluating χ as a query in a Pl database corresponding to X results in a non-empty list of substitutions (or `true`).[1] Also, in the remainder of this subsection, the sets $literals^{+}$, $literals^{-}$, $free$, and $domain$ occur, which were introduced in Chap. 2. More specifically, recall from Sect. 2.3.2 that $literals^{+}(\chi)$ is the set of positive literals in χ, that $literals^{-}(\chi)$ is the set of negative literals in χ, that $free(\chi)$ is the set of free variables in χ, and that $domain(\theta)$ is the set of variables bound by θ.

[1] If X contains only facts and χ is a fact, then the $\models$ relation can be thought of as a unification relation, i.e. stating that χ is unifiable with a fact in X.

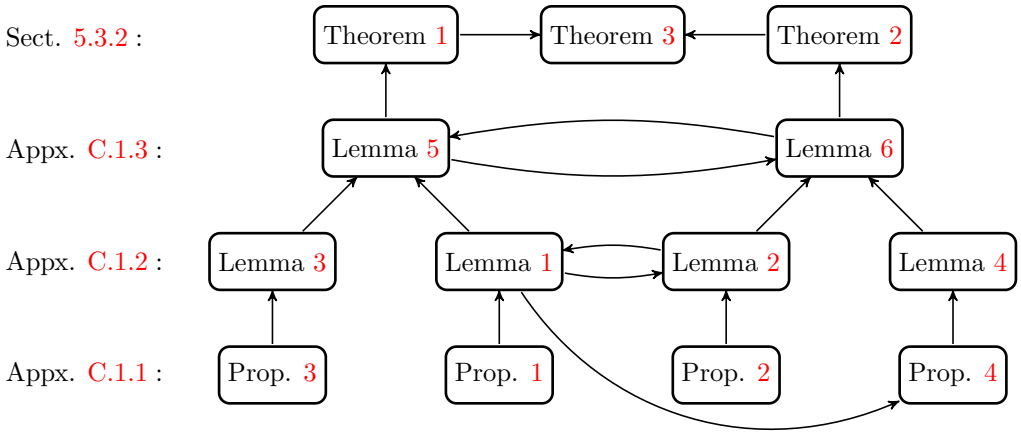


Figure C.1: Schematic overview of how propositions, lemmas, and theorems discussed in this section depend on each other.

Add_Σ, Del_Σ propositions

We start with two propositions that concern Add_Σ, i.e. Prop. 1, and Del_Σ, i.e. Prop. 2. In essence, both propositions are very similar with respect to what they express, as well as with respect to how they are proven. We first discuss their meaning and relevance to the remainder. Let $\boldsymbol{\tau}(\rho)$ be a transition class such that $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$. Additionally, let $\tau \in \boldsymbol{\tau}(\rho)$ such that $\Delta(t) = \langle\langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma\rangle, \langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma\rangle\rangle$ for all $t \in \tau$. Proposition 1 states that all facts that are derivable from Add_Σ are also derivable from $\mathsf{ApproxAdd}_\Sigma$. Similarly, Prop. 2 states that all facts that are derivable from Del_Σ are also derivable from $\mathsf{ApproxDel}_\Sigma$.

The relevance of these propositions is that they allow us to use the approximate $\mathsf{ApproxAdd}_\Sigma$ (and $\mathsf{ApproxDel}_\Sigma$) instead of the precise Add_Σ (and Del_Σ) to compute approximate write sets. This is important, because we can compute $\mathsf{ApproxAdd}_\Sigma$ (and $\mathsf{ApproxDel}_\Sigma$), but not Add_Σ (and Del_Σ) before the transition system is generated. Note that "approximation" in this setting is not exactly a subset relation between Add_Σ and $\mathsf{ApproxAdd}_\Sigma$ (or Del_Σ and $\mathsf{ApproxDel}_\Sigma$). The reason is the possible occurrence of PL variables in Add_Σ (or Del_Σ). For example, $\{\texttt{foo(bar)}\} \not\subseteq \{\texttt{foo(X)}\}$, but the facts that are derivable from $\{\texttt{foo(bar)}\}$ are a subset of the facts that are derivable from $\{\texttt{foo(X)}\}$, which is the only desired property here.

Proposition 1. *Let $\boldsymbol{\tau}(\rho)$ be a transition class such that $\rho =$* **if** ψ **then** α *and* $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$.
Additionally, let $\tau \in \boldsymbol{\tau}(\rho)$ such that $\Delta(t) = \langle\langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma\rangle, \langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma\rangle\rangle$ for all $t \in \tau$.
Additionally, let χ'' be a fact such that $\mathsf{Add}_\Sigma \models \chi''$.
Then:

$$\mathsf{ApproxAdd}_\Sigma \models \chi''$$

Proof. Because $\mathsf{Add}_\Sigma \models \chi''$ and χ'' is a fact, $\mathsf{Add}_\Sigma \neq \emptyset$.
Then, by the mental state transformer, $\alpha = \langle \chi_{\texttt{pre}}, \chi_{\texttt{post}}\rangle$ such that there exists a substitution θ' such that $\mathit{domain}(\theta') = \mathit{free}(\chi_{\texttt{post}})$ and $\mathsf{Add}_\Sigma = \mathit{literals}^+(\chi_{\texttt{post}}\theta')$.[2]
Then, because $\mathsf{ApproxAdd}_\Sigma = \mathit{literals}^+(\chi_{\texttt{post}})$ by Def. 11, for all $\chi \in \mathsf{Add}_\Sigma$, there exists a $\chi' \in \mathsf{ApproxAdd}_\Sigma$ such that $\chi = \chi'\theta'$
Then, because Add_Σ contains only ground facts, $\mathsf{ApproxAdd}_\Sigma \models \chi''$.
This establishes the proposition. □

[2] Beliefs can only be added to the belief base by means of user-defined actions. The additions are ground instantiations of positive literals occurring in the postcondition.

Proposition 2. *Let* $\boldsymbol{\tau}(\rho)$ *be a transition class such that* $\rho =$ **if** ψ **then** α *and* $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$.
Additionally, let $\tau \in \boldsymbol{\tau}(\rho)$ *such that* $\Delta(t) = \langle\langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma\rangle, \langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma\rangle\rangle$ *for all* $t \in \tau$.
Additionally, let χ'' *be a fact such that* $\mathsf{Del}_\Sigma \models \chi''$.
Then:

$$\mathsf{ApproxDel}_\Sigma \models \chi''$$

Proof. Then, because $\mathsf{Del}_\Sigma \models \chi''$ and χ'' is a fact, $\mathsf{Del}_\Sigma \neq \emptyset$.
Then, by the mental state transformer, $\alpha = \langle \chi_{\mathtt{pre}}, \chi_{\mathtt{post}}\rangle$ such that there exists a substitution θ' such that $\mathit{domain}(\theta') = \mathit{free}(\chi_{\mathtt{post}})$ and $\mathsf{Del}_\Sigma = \{\chi \in \mathcal{L} \mid \mathtt{not}(\chi) \in \mathit{literals}^-(\chi_{\mathtt{post}}\theta')\}$.[3]
Then, because $\mathsf{ApproxDel}_\Sigma = \{\chi \mid \mathtt{not}(\chi) \in \mathit{literals}^-(\chi_{\mathtt{post}})\}$ by Def. 11, for all $\chi \in \mathsf{Add}_\Sigma$, there exists a $\chi' \in \mathsf{ApproxDel}_\Sigma$ such that $\chi = \chi'\theta'$
Then, because Del_Σ contains only ground facts, $\mathsf{ApproxDel}_\Sigma \models \chi''$.
This establishes the proposition. □

Add_Γ, Del_Γ propositions

The previous two propositions concerned operations on the belief base only, and as such were fairly easy to prove. The next two propositions concern the goal base, and are a bit more elaborate. In essence, however, Props. 3,4 express similar claims as Props. 1,2. That is, let $\boldsymbol{\tau}(\rho)$ be a transition class such that $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$. Additionally, let $\tau \in \boldsymbol{\tau}(\rho)$ such that $\Delta(t) = \langle\langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma\rangle, \langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma\rangle\rangle$ for all $t \in \tau$. Proposition 3 states that all facts that are derivable from a goal in Add_Γ are also derivable from a goal in $\mathsf{ApproxAdd}_\Gamma$. Likewise, Prop. 4 states that all facts that are derivable from a goal in Del_Γ are also derivable from a goal in $\mathsf{ApproxDel}_\Gamma$. As above, note that "approximation" is not exactly a subset relation (for the same reason).

We start with the simpler proposition, which concerns $\mathsf{ApproxAdd}_\Gamma$. Recall from Sect. 2.3.1 that for notational convenience, we regard a conjunction of facts (e.g. a goal template) as a set containing all conjuncts (and nothing else).

Proposition 3. *Let* $\boldsymbol{\tau}(\rho)$ *be a transition class such that* $\rho =$ **if** ψ **then** α *and* $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$.
Additionally, let $\tau \in \boldsymbol{\tau}(\rho)$ *such that* $\Delta(t) = \langle\langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma\rangle, \langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma\rangle\rangle$ *for all* $t \in \tau$.
Additionally, let χ'' *be a fact and let* $\gamma \in \mathsf{Add}_\Gamma$ *such that* $\gamma \models \chi''$.
Then:

$$\text{there exists a } \gamma' \in \mathsf{ApproxAdd}_\Gamma \text{ such that } \gamma' \models \chi''$$

Proof. By the mental state transformer, $\alpha = \mathbf{adopt}(\chi_\gamma)$ such that there exists a substitution θ' such that $\mathit{domain}(\theta') = \mathit{free}(\chi_\gamma)$ and $\gamma = \mathit{literals}^+(\chi_\gamma\theta)$.[4]
Then, because $\gamma'' \in \mathsf{ApproxAdd}_\Gamma$ for $\gamma'' = \mathit{literals}^+(\chi_\gamma)$ by Def. 13, there exists a $\gamma' \in \mathsf{ApproxAdd}_\Gamma$ (namely γ'') such that for all $\chi \in \gamma$, there exists a $\chi' \in \gamma'$ such that $\chi = \chi'\theta'$.
Then, because γ contains ground facts only, $\gamma' \models \chi''$.
This establishes the proposition. □

The last proposition of this subsection is the toughest. The reason is twofold. On the one hand, in contrast to the previous three propositions, deletions of goals from the goal base can be brought about by two types of actions (namely user-defined and **drop**) rather than a single. On the other hand, goals that are deleted do not occur explicitly in the code by which action rules and specifications are defined, which is why these are approximated by means of *Goals*.

[3] Beliefs can only be deleted from the belief base by means of user-defined actions. The deletions are ground instantiations of negative literals occurring in the postcondition, whose `not` operator has been stripped off.

[4] Goals can only be added to the goal base by means of **adopt** actions. The addition is a set of ground instantiations of positive literals occurring in the **adopt** action's argument.

We sketch the proof as follows. First, it is established that if a goal γ is removed from the goal base, then there exists a corresponding goal template in *Goals*. Because γ contains ground facts only, at least the same facts that can be derived from it can also be derived from this template. Consequently, we only need to prove that this template is also in $\mathsf{ApproxDel}_\Gamma$. This is done with case distinction on the type of action. In case of a user-defined action, the proof is based on the fact that for a goal to become achieved, at least one of its conjuncts must become derivable from the belief and knowledge bases combined (otherwise, the goal was already achieved before). In case of a **drop** action, the proof is more elaborate. Specifically, this case is proven by induction on the structure of the **drop** action's argument. Moreover, the base case of this induction is proven with an inner inductive proof on the number of rule applications required to derive the **drop** action's argument from the instantiated goal template.

We remark that in the proof of Prop. 4 there is a reference to Lemma 1, which we treat in Appx. C.1.2. We could have first presented Lemma 1, but we favour the current order, because propositions and lemmas are more neatly "categorised" as such. Also, recall from Sect. 2.3.1 that for notational convenience, we regard a conjunction of facts (e.g. a goal template) as a set containing all conjuncts (and nothing else).

Proposition 4. *Let* $\boldsymbol{\tau}(\rho)$ *be a transition class such that* $\rho = \textbf{if } \psi \textbf{ then } \alpha$ *and* $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$.
Additionally, let $\tau \in \boldsymbol{\tau}(\rho)$ *such that* $\Delta(t) = \langle\langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma\rangle, \langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma\rangle\rangle$ *for all* $t \in \tau$.
Additionally, let χ'' *be a fact and let* $\gamma \in \mathsf{Del}_\Gamma$ *such that* $\gamma \models \chi''$.
Then:

$$\text{there exists a } \gamma' \in \mathsf{ApproxDel}_\Gamma \text{ such that } \gamma' \models \chi''$$

Proof. Let $t \in \tau$ in which $t = \langle \mu, \mu' \rangle$ and $\mu = \langle K, \Sigma, \Gamma \rangle$ and $\mu' = \langle K, \Sigma', \Gamma' \rangle$ such that $\mathsf{Del}_\Gamma = \Gamma \setminus \Gamma'$.
By definition of *Goals* in Def. 14, there exist a $\gamma' \in \mathit{Goals}$ and substitution θ such that for all $\chi \in \gamma$, there exists a $\chi' \in \gamma'$ such that $\chi = \chi'\theta$.[5]
Then, because γ contains ground facts only, $\gamma' \models \chi$.[6]
Also, by the mental state transformer, $\alpha = \langle \chi_{\mathtt{pre}}, \chi_{\mathtt{post}} \rangle$ or $\alpha = \textbf{drop}(\chi_\gamma)$.[7]
We proceed with case distinction.

- Suppose $\alpha = \langle \chi_{\mathtt{pre}}, \chi_{\mathtt{post}} \rangle$.
  Then, by the mental state transformer, there exists a $\chi \in \gamma$ such that $K \cup \Sigma \not\models \chi$ and $K \cup \Sigma' \models \chi$.[8]
  Then, by definition of γ' above, there exists a $\chi' \in \gamma'$ such that $\chi = \chi'\theta$.
  Then, because $K \cup \Sigma \not\models \chi$ and $K \cup \Sigma' \models \chi$, $K \cup \Sigma \not\models \chi'$ and $K \cup \Sigma' \models \chi'$.
  Then, by Lemma 1, $der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi')$.
  Then, because $\chi' \in \gamma'$ and by definition of $\mathsf{ApproxDel}_\Gamma$ in Def. 15, $\gamma' \in \mathsf{ApproxDel}_\Gamma$.

- Suppose $\alpha = \textbf{drop}(\chi_d)$.
  Then, by the mental state transformer, $K \cup \gamma \models \chi_d$.
  We proceed with induction on the structure of χ_d.

  Base χ_d is a fact.
  We proceed with induction on the number of rule applications l of rules in the knowledge base K minimally required to derive χ_d from $K \cup \gamma$.

[5] The set *Goals* contains the goals in the initial goal base, as well as all the uninstantiated goal templates that occur in the code as the argument of **adopt** actions. A goal in the goal base at runtime is either an initial goal, or a ground instantiation of a goal template.
[6] Note that at this point in the proof, we only need to establish that γ' is an element of $\mathsf{ApproxDel}_\Gamma$.
[7] Goals can only be deleted from the goal base by means of user-defined or **drop** actions. In the former case, the performance of such an action causes a goal to become achieved, whereas in the latter case, the **drop** action's argument is derivable from a goal (combined with knowledge).
[8] For a goal to become achieved by performance of a user-defined action, at least one conjunct of this goal must become derivable (otherwise, the goal was already achieved before).

Base $l = 0$.
Because $l = 0$, $\gamma \models \chi_d$.[9]
Then, because γ contains ground facts only, $\gamma' \models \chi_d$.
Then, by definition of der^+ in Def. 16, $der^+(K, \gamma', \{\texttt{true}\}, \chi_d)$.

Hypothesis Let IH' be the hypothesis that $der^+(K, \gamma', \{\texttt{true}\}, \chi_d)$ when $l = k$ rule applications are required.

Step $l = k + 1$.
Because $l > 0$, $\gamma \not\models \chi_d$.
Then, because $K \cup \gamma \models \chi_d$, there exist a substitution θ and an $r \in K$ in which $r = \chi_h \texttt{ :- } \chi_b$ such that $\chi_h\theta = \chi_d\theta$ and $K \cup \gamma \models \chi_b\theta$.
Then, by IH', $der^+(K, \gamma', \{\texttt{true}\}, \chi_b\theta)$.[10]
Then, by definition of der^+ in Def. 16, $der^+(K, \gamma', \{\texttt{true}\}, \chi_d)$.

Thus, $der^+(K, \gamma', \{\texttt{true}\}, \chi_d)$ when χ_d is a fact.

Hypothesis Let IH be the hypothesis that $der^+(K, \gamma', \{\texttt{true}\}, \chi_d)$ for all arguments χ_i of a term $\chi_d = \chi_0(\chi_1, \ldots, \chi_n)$ in which $1 \leq i \leq n$.

Step χ_d is not a fact.
We proceed with case distinction on χ_d.

- Suppose $\chi_d = \texttt{not}(\chi')$ such that $K \cup \gamma \models \texttt{not}(\chi')$.
  Then, by definition of der^+ in Def. 16, $der^+(K, \gamma', \{\texttt{true}\}, \chi)$.[11]
- Suppose $\chi_d = \chi_1\texttt{,}\chi_2$ such that $K \cup \gamma \models \chi_1\texttt{,}\chi_2$.
  Then, by definition of $\texttt{,}$, $K \cup \gamma \models \chi_1$ or $K \cup \gamma \models \chi_2$.
  Then, by IH, $der^+(K, \gamma', \{\texttt{true}\}, \chi_1)$ or $der^+(K, \gamma', \{\texttt{true}\}, \chi_2)$.
  Then, by definition of der^+ in Def. 16, $der^+(K, \gamma', \{\texttt{true}\}, \chi)$.

Thus, $der^+(K, \gamma', \{\texttt{true}\}, \chi_d)$.
Then, by definition of $\mathsf{ApproxDel}_\Gamma$ in Def. 15, $\gamma' \in \mathsf{ApproxDel}_\Gamma$.

Then, because $\gamma' \models \chi$ and $\gamma' \in \mathsf{ApproxDel}_\Gamma$, there exists a $\gamma' \in \mathsf{ApproxDel}_\Gamma$ such that $\gamma' \models \chi$.
This establishes the proposition. □

C.1.2 *der* Lemmas

Before continuing, let us recapitulate the previous. We have introduce Props. 1,2,3,4 which can be regarded as the foundation on which the other lemmas and theorems are built. That is, if we consider Fig. C.1, we have established the lowest layer at this point. Next, in this section, we discuss Lemmas 1,2,3,4, starting with the former two lemmas, and ending with the latter two.

The first lemma that we treat, i.e. Lemma 1, informally states the following. Let $\boldsymbol{\tau}(\rho)$ be a transition class such that $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle\mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle\mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$, and let K be the knowledge base. Then, Lemma 1 states that if the execution of a transition in $\boldsymbol{\tau}(\rho)$ can make a term χ derivable from the knowledge and belief bases combined, $der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$ is true. A consequence of this lemma is that to determine whether there exists a transition in $\boldsymbol{\tau}(\rho)$ that can change the truth value of a mental literal $\mathbf{bel}(\chi)$ from false to true, we can compute $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho))$ and check $der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$. This line of reasoning is embedded in the definition of Der^+ in Def. 18, and as such, Lemma 1 is an important building block for the proof of Lemma 5 in Appx. C.1.3, which concerns Der^+.

The structure of the proof of this lemma is very similar to the structure of the proof of the **drop** case in Prop. 4. That is, Lemma 1 is proven by induction on the structure of χ, and the base case of this induction is proven with an inner inductive proof on the number of rule applications

[9] If no rule applications are necessary to derive χ_d from $K \cup \gamma$, then χ_d is derivable from γ alone.
[10] IH' is applicable because $\chi_b\theta$ is derivable from $K \cup \gamma$ with k applications of rules in K.
[11] Note that we do not look at χ' in this case. That is, by using the $\{\texttt{true}\}$ wild card, it is assumed without further analysis that $\texttt{not}(\chi')$ can become derivable.

of rules in the knowledge base that is minimally required to derive χ. The base of this inner induction is that no applications are necessary. In such case, χ becomes derivable exclusively due to additions to the belief base. The hypothesis of the inner induction is that the lemma is true when k applications are required, and in the inductive step we prove that the lemma is also true for $k+1$.

Lemma 1. *Let* $\boldsymbol{\tau}(\rho)$ *be a transition class such that* $\rho =$ **if** ψ **then** α *and* $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$.
Additionally, let $\tau \in \boldsymbol{\tau}(\rho)$*, and let* $t \in \tau$ *in which* $t = \langle\langle K, \Sigma, \Gamma\rangle, \langle K, \Sigma', \Gamma'\rangle\rangle$ *such that* $\Delta(t) = \langle\langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma\rangle, \langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma\rangle\rangle$.
Additionally, let χ *be a term such that* $K \cup \Sigma \not\models \chi$ *and* $K \cup \Sigma' \models \chi$.
Then:

$$der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$$

Proof. By induction on the structure of χ.

Base χ is a fact.
We proceed with induction on the number of rule applications l of rules in the knowledge base K minimally required to derive χ from $K \cup \Sigma'$.

Base $l = 0$.
Because $l = 0$, $\Sigma \not\models \chi$ and $\Sigma' \models \chi$.[12]
Then, because Σ and Σ' contain facts only, $\Sigma' \setminus \Sigma \models \chi$.
Then, because $\mathsf{Add}_\Sigma = \Sigma' \setminus \Sigma$ by Def. 5, $\mathsf{Add}_\Sigma \models \chi$.
Then, by Prop. 1, $\mathsf{ApproxAdd}_\Sigma \models \chi$.
Then, by definition of der^+ in Def. 16, $der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$.

Hypothesis Let Ih' be the hypothesis that $der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$ when $l = k$ rule applications are required.

Step $l = k + 1$.
Because $l > 0$, $\Sigma' \not\models \chi$.
Then, because $K \cup \Sigma' \models \chi$, there exist a substitution θ and an $r \in K$ in which $r = \chi_h \texttt{ :- } \chi_b$ such that $\chi_h\theta = \chi\theta$ and $K \cup \Sigma \not\models \chi_b\theta$ and $K \cup \Sigma' \models \chi_b\theta$.[13]
Then, by Ih', $der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi_b\theta)$.[14]
Then, by definition of der^+ in Def. 16, $der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$.

Thus, $der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$ when χ is a fact.

Hypothesis Let Ih be the hypothesis that $der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi_i)$ for all arguments χ_i of a term $\chi = \chi_0(\chi_1, \ldots, \chi_n)$ in which $1 \leq i \leq n$.

Step χ is not a fact.
We proceed with case distinction on χ.

- Suppose $\chi = \texttt{not}(\chi')$ such that $K \cup \Sigma \not\models \texttt{not}(\chi')$ and $K \cup \Sigma' \models \texttt{not}(\chi')$.
  Then, by definition of not, $K \cup \Sigma \models \chi'$ and $K \cup \Sigma' \not\models \chi'$.
  Then, by Lemma 2, $der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi')$.[15]
  Then, by definition of der^+ in Def. 16, $der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$.

[12] If no rule applications are necessary to derive χ from $K \cup \Sigma'$, then χ is derivable from Σ' alone. Also, because $K \cup \Sigma \not\models \chi$, χ is also not derivable from Σ alone.
[13] $K \cup \Sigma \not\models \chi_b\theta$ because otherwise, $\chi_h\theta$ would already have been derivable, i.e. $K \cup \Sigma \models \chi_h\theta$ such that χ would already have been derivable, i.e. $K \cup \Sigma \models \chi$, which is not the case.
[14] Ih' is applicable because $\chi_b\theta$ is derivable from $K \cup \Sigma'$ with k applications of rules in K.
[15] We refer here to Lemma 2. However, in the proof of that lemma, we refer to *this* lemma, which might seem to yield a circular proof. This is not the case. The inductive applications of the two proofs eventually "halt" because the inner structure of Pl terms is finite (note that we apply Lemma 2 on the argument of a term). The two lemmas could be merged, but for the sake of readability, we did not do this.

- Suppose $\chi = \chi_1 \text{,} \chi_2$ such that $K \cup \Sigma \not\models \chi_1 \text{,} \chi_2$ and $K \cup \Sigma' \models \chi_1 \text{,} \chi_2$.
  Then, by definition of ,, $K \cup \Sigma \not\models \chi_1$ and $K \cup \Sigma' \models \chi_1$, or $K \cup \Sigma \not\models \chi_2$ and $K \cup \Sigma' \models \chi_2$.
  Then, by IH, $der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi_1)$ or $der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi_2)$.
  Then, by definition of der^+ in Def. 16, $der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$.

This establishes the lemma. □

The following lemma is very similar to the previous. Lemma 2 states that if the execution of a transition in $\boldsymbol{\tau}(\rho)$ can make a term χ underivable from the knowledge and belief bases combined, $der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$ is true. A consequence of this lemma is that to determine whether there exists a transition in $\boldsymbol{\tau}(\rho)$ that can change the truth value of a mental literal $\mathbf{bel}(\chi)$ from true to false, we can compute $\mathsf{ApproxΔ}(\boldsymbol{\tau}(\rho))$ and determine whether $der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$. This line of reasoning is embedded in the definition of Der^- in Def. 19, and as such, Lemma 2 is an important building block for the proof of Lemma 6 in Appx. C.1.3, which concerns Der^-. The structure of the proof is identical to the structure of the proof for Lemma 1.

Lemma 2. *Let $\boldsymbol{\tau}(\rho)$ be a transition class such that $\rho = \mathbf{if}\ \psi\ \mathbf{then}\ \alpha$ and $\mathsf{ApproxΔ}(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$.*
Additionally, let $\tau \in \boldsymbol{\tau}(\rho)$, and let $t \in \tau$ in which $t = \langle\langle K, \Sigma, \Gamma\rangle, \langle K, \Sigma', \Gamma'\rangle\rangle$ such that $\Delta(t) = \langle\langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma\rangle, \langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma\rangle\rangle$.
Additionally, let χ be a term such that $K \cup \Sigma \models \chi$ and $K \cup \Sigma' \not\models \chi$.
Then:

$$der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$$

Proof. By induction on the structure of χ.

Base χ is a fact.
We proceed with induction on the number of rule applications l of rules in the knowledge base K minimally required to derive χ from $K \cup \Sigma$.

Base $l = 0$.
Because $l = 0$, $\Sigma \models \chi$ and $\Sigma' \not\models \chi$.[16]
Then, because Σ and Σ' contain facts only, $\Sigma \setminus \Sigma' \models \chi$.
Then, because $\mathsf{Del}_\Sigma = \Sigma \setminus \Sigma'$ by Def. 5, $\mathsf{Del}_\Sigma \models \chi$.
Then, by Prop. 2, $\mathsf{ApproxDel}_\Sigma \models \chi$.
Then, by definition of der^- in Def. 17, $der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$.

Hypothesis Let IH' be the hypothesis that $der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$ when $l = k$ rule applications are required.

Step $l = k + 1$ Because $l > 0$, $\Sigma \not\models \chi$.
Then, because $K \cup \Sigma \models \chi$, there exist a substitution θ and an $r \in K$ in which $r = \chi_h \text{ :- } \chi_b$ such that $\chi_h\theta = \chi\theta$ and $K \cup \Sigma \models \chi_b\theta$ and $K \cup \Sigma' \not\models \chi_b\theta$.[17]
Then, by IH', $der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi_b\theta)$.[18]
Then, by definition of der^- in Def. 17, $der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$.

Thus, $der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$ when χ is a fact.

Hypothesis Let IH be the hypothesis that $der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi_i)$ for all arguments χ_i of a term $\chi = \chi_0(\chi_1, \ldots, \chi_n)$ in which $1 \leq i \leq n$.

[16] If no rule applications are necessary to derive χ from $K \cup \Sigma$, then χ is derivable from Σ alone. Also, because $K \cup \Sigma' \not\models \chi$, χ is also not derivable from Σ' alone.
[17] $K \cup \Sigma' \not\models \chi_b\theta$ because otherwise, $\chi_h\theta$ is derivable, i.e. $K \cup \Sigma' \models \chi_h\theta$ such that χ is derivable, i.e. $K \cup \Sigma' \models \chi$, which is not the case.
[18] IH' is applicable because $\chi_b\theta$ is derivable from $K \cup \Sigma$ with k applications of rules in K.

Step χ is not a fact.
We proceed with case distinction on χ.

- Suppose $\chi = \texttt{not}(\chi')$ such that $K \cup \Sigma \models \texttt{not}(\chi')$ and $K \cup \Sigma' \not\models \texttt{not}(\chi')$.
  Then, by definition of `not`, $K \cup \Sigma \not\models \chi'$ and $K \cup \Sigma' \models \chi'$.
  Then, by Lemma 1, $der^+(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi')$.[19]
  Then, by definition of der^- in Def. 17, $der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$.
- Suppose $\chi = \chi_1\texttt{,}\chi_2$ such that $K \cup \Sigma \models \chi_1\texttt{,}\chi_2$ and $K \cup \Sigma' \not\models \chi_1\texttt{,}\chi_2$.
  Then, by definition of `,`, $K \cup \Sigma \models \chi_1$ and $K \cup \Sigma' \not\models \chi_1$, or $K \cup \Sigma \models \chi_2$ and $K \cup \Sigma' \not\models \chi_2$.
  Then, by IH, $der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi_1)$ or $der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi_2)$.
  Then, by definition of der^- in Def. 17, $der^-(K, \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxDel}_\Sigma, \chi)$.

This establishes the lemma. □

The previously discussed lemmas concern the belief base. Next, we present two lemmas that concern the goal base. Because the two lemmas are very alike, we informally outline both of them simultaneously before presenting either of them. Let $\boldsymbol{\tau}(\rho)$ be a transition class such that $\mathsf{ApproxA}(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$. Lemma 3 states that if a PL term χ is derivable from a goal (combined with knowledge) that is added to the goal base by a transition in $\boldsymbol{\tau}(\rho)$, then there exists a goal γ'' in $\mathsf{ApproxAdd}_\Gamma$ such that $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$ is true. Similarly, Lemma 3 states that if χ is derivable from a goal (combined with knowledge) that is deleted from the goal base by a transition in $\boldsymbol{\tau}(\rho)$, then there exists a goal γ'' in $\mathsf{ApproxDel}_\Gamma$ such that $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$ is true.

A consequence of Lemma 3 is that to determine whether there exists a transition in $\boldsymbol{\tau}(\rho)$ that can change the truth value of a mental literal $\mathbf{goal}(\chi)$ from false to true, we can compute $\mathsf{ApproxA}(\boldsymbol{\tau}(\rho))$ and determine whether there exists a goal γ'' in $\mathsf{ApproxAdd}_\Gamma$ such that $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$. This line of reasoning is embedded in the definition of Der^+ in Def. 18, and as such, Lemma 3 is an important building block for the proof of Lemma 5 in Appx. C.1.3, which concerns Der^+. Similarly, a consequence of Lemma 4 is that to determine whether there exists a transition in $\boldsymbol{\tau}(\rho)$ that can change the truth value of a mental literal $\mathbf{goal}(\chi)$ from true to false, we can compute $\mathsf{ApproxA}(\boldsymbol{\tau}(\rho))$ and determine whether there exists a goal γ'' in $\mathsf{ApproxDel}_\Gamma$ such that $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$. This line of reasoning is embedded in the definition of Der^- in Def. 19, and as such, Lemma 4 is an important building block for the proof of Lemma 6 in Appx. C.1.3, which concerns Der^-.

The proofs of both lemmas are similar to each other, and their structure is very much alike the structure of the proofs for Lemmas 1,2: an outer proof by induction on the structure of χ and combined with an inner inductive proof on the number of rule applications required to derive χ from a goal (combined with the knowledge base). Recall from Sect. 2.3.1 that for notational convenience, we regard a conjunction of facts (e.g. a goal) as a set containing all conjuncts (and nothing else).

Lemma 3. *Let $\boldsymbol{\tau}(\rho)$ be a transition class such that $\rho =$ **if** ψ **then** α and $\mathsf{ApproxA}(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$.*
Additionally, let $\tau \in \boldsymbol{\tau}(\rho)$, and let $t \in \tau$ in which $t = \langle\langle K, \Sigma, \Gamma\rangle, \langle K, \Sigma', \Gamma'\rangle\rangle$ such that $\Delta(t) = \langle\langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma\rangle, \langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma\rangle\rangle$.
Additionally, let χ be a term such that $K \cup \gamma \not\models \chi$ for all $\gamma \in \Gamma$ and there exists a $\gamma' \in \Gamma'$ such that $K \cup \gamma' \models \chi$.
Then:

$$\text{there exists a } \gamma'' \in \mathsf{ApproxAdd}_\Gamma \text{ such that } der^+(K, \gamma'', \{\texttt{true}\}, \chi)$$

[19] We refer here to Lemma 1. However, in the proof of that lemma, we refer to *this* lemma, which might seem to yield a circular proof. This is not the case. The inductive applications of the two proofs eventually "halt" because the inner structure of PL terms is finite (note that we apply Lemma 1 on the argument of a term). The two lemmas could be merged, but for the sake of readability, we did not do this.

Proof. By induction on the structure of χ.

Base χ is a fact.
We proceed with induction on the number of rule applications l of rules in the knowledge base K minimally required to derive χ from $K \cup \gamma'$.

Base $l = 0$.
Because $l = 0$, $\gamma \not\models \chi$ for all $\gamma \in \Gamma$ and $\gamma' \models \chi$.[20]
Then, because $\gamma' \notin \Gamma$ and by definition of Δ in Def. 5, $\gamma' \in \mathsf{Add}_\Gamma$.
Then, because $\gamma' \models \chi$ and by Prop. 3, there exists a $\gamma'' \in \mathsf{ApproxAdd}_\Gamma$ such that $\gamma'' \models \chi$.
Then, by definition of der^+ in Def. 16, $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$.

Hypothesis Let IH' be the hypothesis that $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$ when $l = k$ rule applications are required.

Step $l = k + 1$.
Because $l > 0$, $\gamma' \not\models \chi$.
Then, because $K \cup \gamma' \models \chi$, there exist a substitution θ and an $r \in K$ in which $r = \chi_h \texttt{ :- } \chi_b$ such that $\chi_h\theta = \chi\theta$ and $K \cup \gamma \not\models \chi_b\theta$ for all $\gamma \in \Gamma$ and $K \cup \gamma' \models \chi_b\theta$.[21]
Then, by IH', there exists a $\gamma'' \in \mathsf{ApproxAdd}_\Gamma$ such that $der^+(K, \gamma'', \{\texttt{true}\}, \chi_b\theta)$.[22]
Then, by definition of der^+ in Def. 16, $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$.

Thus, there exists a $\gamma'' \in \mathsf{ApproxAdd}_\Gamma$ such that $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$ when χ is a fact.

Hypothesis Let IH be the hypothesis that there exists a $\gamma'' \in \mathsf{ApproxAdd}_\Gamma$ such that $der^+(K, \gamma'', \{\texttt{true}\}, \chi_i)$ for all arguments χ_i of a term $\chi = \chi_0(\chi_1, \ldots, \chi_n)$ in which $1 \leq i \leq n$.

Step χ is not a fact.
We proceed with case distinction on χ.

- Suppose $\chi = \texttt{not}(\chi')$ such that $K \cup \gamma \not\models \texttt{not}(\chi')$ for all $\gamma \in \Gamma$ and $K \cup \gamma' \models \texttt{not}(\chi')$.
  Then, by definition of der^+ in Def. 16, $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$ for all $\gamma'' \in \mathsf{ApproxAdd}_\Gamma$.[23]
- Suppose $\chi = \chi_1\texttt{,}\chi_2$ such that $K \cup \gamma \not\models \chi_1\texttt{,}\chi_2$ for all $\gamma \in \Gamma$ and $K \cup \gamma' \models \chi_1\texttt{,}\chi_2$.
  Then, by definition of $\texttt{,}$, $K \cup \gamma \not\models \chi_1$ for all $\gamma \in \Gamma$ and $K \cup \gamma' \models \chi_1$, or $K \cup \gamma \not\models \chi_2$ for all $\gamma \in \Gamma$ and $K \cup \gamma' \models \chi_2$.
  Then, by IH, there exists a $\gamma'' \in \mathsf{ApproxAdd}_\Gamma$ such that $der^+(K, \gamma'', \{\texttt{true}\}, \chi_1)$ or $der^+(K, \gamma'', \{\texttt{true}\}, \chi_2)$.
  Then, by definition of der^+ in Def. 16, $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$.

This establishes the lemma. □

Lemma 4. *Let* $\boldsymbol{\tau}(\rho)$ *be a transition class such that* $\rho =$ **if** ψ **then** α *and* $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$.
Additionally, let $\tau \in \boldsymbol{\tau}(\rho)$, *and let* $t \in \tau$ *in which* $t = \langle\langle K, \Sigma, \Gamma\rangle, \langle K, \Sigma', \Gamma'\rangle\rangle$ *such that* $\Delta(t) = \langle\langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma\rangle, \langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma\rangle\rangle$.
Additionally, let χ *be a term such that there exists a* $\gamma \in \Gamma$ *such that* $K \cup \gamma \models \chi$ *and* $K \cup \gamma' \not\models \chi$ *for all* $\gamma' \in \Gamma$.
Then:

$$\text{there exists a } \gamma'' \in \mathsf{ApproxDel}_\Gamma \text{ such that } der^+(K, \gamma'', \{\texttt{true}\}, \chi)$$

[20] If no rule applications are necessary to derive χ from $K \cup \gamma'$, then χ is derivable from γ' alone. Also, because $K \cup \gamma \not\models \chi$ for all $\gamma \in \Gamma$, χ is also not derivable from each such γ alone.
[21] $K \cup \gamma \not\models \chi_b\theta$ for all $\gamma \in \Gamma$ because otherwise, $\chi_h\theta$ would be derivable from at least one such goal γ, i.e. $K \cup \gamma \models \chi_h\theta$ such that χ would already have been derivable, i.e. $K \cup \gamma \models \chi$, which is not the case.
[22] IH' is applicable because $\chi_b\theta$ is derivable from $K \cup \gamma'$ with k applications of rules in K.
[23] Note that we do not look at χ' in this case. That is, by using the $\{\texttt{true}\}$ wild card, it is assumed without further analysis that $\texttt{not}(\chi')$ can become derivable from a goal.

Proof. By induction on the structure of χ.

Base χ is a fact.
We proceed with induction on the number of rule applications l of rules in the knowledge base K minimally required to derive χ from $K \cup \gamma$.

Base $l = 0$.
Because $l = 0$, $\gamma \models \chi$ and $\gamma' \not\models \chi$ for all $\gamma' \in \Gamma$.[24]
Then, because $\gamma \notin \Gamma'$ and by definition of Δ in Def. 5, $\gamma \in \mathsf{Del}_\Gamma$.
Then, because $\gamma \models \chi$ and by Prop. 4, there exists a $\gamma'' \in \mathsf{ApproxDel}_\Gamma$ such that $\gamma'' \models \chi$.
Then, by definition of der^+ in Def. 16, $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$.

Hypothesis Let IH' be the hypothesis that $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$ when $l = k$ rule applications are required.

Step $l = k + 1$.
Because $l > 0$, $\gamma \not\models \chi$.
Then, because $K \cup \gamma \models \chi$, there exist a substitution θ and an $r \in K$ in which $r = \chi_h \texttt{ :- } \chi_b$ such that $\chi_h\theta = \chi\theta$ and $K \cup \gamma \models \chi_b\theta$ and $K \cup \gamma' \not\models \chi_b\theta$ for all $\gamma' \in \Gamma'$.[25]
Then, by IH', there exists a $\gamma'' \in \mathsf{ApproxDel}_\Gamma$ such that $der^+(K, \gamma'', \{\texttt{true}\}, \chi_b\theta)$.[26]
Then, by definition of der^+ in Def. 16, $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$.

Thus, there exists a $\gamma'' \in \mathsf{ApproxDel}_\Gamma$ such that $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$ when χ is a fact.

Hypothesis Let IH be the hypothesis that there exists a $\gamma'' \in \mathsf{ApproxDel}_\Gamma$ such that $der^+(K, \gamma'', \{\texttt{true}\}, \chi_i)$ for all arguments χ_i of a term $\chi = \chi_0(\chi_1, \ldots, \chi_n)$ in which $1 \leq i \leq n$.

Step χ is not a fact.
We proceed with case distinction on χ.

- Suppose $\chi = \texttt{not}(\chi')$ such that $K \cup \gamma \models \texttt{not}(\chi')$ and $K \cup \gamma' \not\models \texttt{not}(\chi')$ for all $\gamma' \in \Gamma'$.
  Then, by definition of der^+ in Def. 16, $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$ for all $\gamma'' \in \mathsf{ApproxDel}_\Gamma$.[27]
- Suppose $\chi = \chi_1\texttt{,}\chi_2$ such that $K \cup \gamma \models \chi_1\texttt{,}\chi_2$ and $K \cup \gamma' \not\models \chi_1\texttt{,}\chi_2$ for all $\gamma' \in \Gamma'$.
  Then, by definition of $\texttt{,}$, $K \cup \gamma \models \chi_1$ and $K \cup \gamma' \not\models \chi_1$ for all $\gamma' \in \Gamma'$, or $K \cup \gamma \models \chi_2$ and $K \cup \gamma' \not\models \chi_2$ for all $\gamma' \in \Gamma'$.
  Then, by IH, there exists a $\gamma'' \in \mathsf{ApproxDel}_\Gamma$ such that $der^+(K, \gamma'', \{\texttt{true}\}, \chi_1)$ or $der^+(K, \gamma'', \{\texttt{true}\}, \chi_2)$.
  Then, by definition of der^+ in Def. 16, $der^+(K, \gamma'', \{\texttt{true}\}, \chi)$.

This establishes the lemma. □

C.1.3 *Der* Lemmas

Before continuing, let us recapitulate what we have discussed up to now. In the previous subsection, we treated Lemmas 1,2,3,4, whose proofs are built on top of the propositions introduced in Appx. C.1.1. We have now established the lower two layers of Fig. C.1; the toughest is behind us. Next, in this subsection, we introduce Lemmas 5,6, which are the last lemmas needed before we can proof our theorems.

The first lemma, i.e. Lemma 5, states that if a mental state condition ψ becomes true by execution of a transition in a transition class $\boldsymbol{\tau}(\rho)$, then $Der^+(\boldsymbol{\tau}(\rho), \psi)$ is true. As a consequence,

[24] If no rule applications are necessary to derive χ from $K \cup \gamma$, then χ is derivable from γ alone. Also, because $K \cup \gamma' \not\models \chi$ for all $\gamma' \in \Gamma'$, χ is also not derivable from each such γ' alone.
[25] $K \cup \gamma' \not\models \chi_b\theta$ for all $\gamma' \in \Gamma'$ because otherwise, $\chi_h\theta$ is derivable from at least one such goal γ', i.e. $K \cup \gamma' \models \chi_h\theta$ such that χ is derivable, i.e. $K \cup \gamma' \models \chi$, which is not the case.
[26] IH' is applicable because $\chi_b\theta$ is derivable from $K \cup \gamma$ with k applications of rules in K.
[27] Note that we do not look at χ' in this case. That is, by using the $\{\texttt{true}\}$ wild card, it is assumed without further analysis that $\texttt{not}(\chi')$ can become derivable from a goal.

to establish whether ψ might become true by execution of a transition in $\boldsymbol{\tau}(\rho)$, we may determine whether $Der^{+}(\boldsymbol{\tau}(\rho), \psi)$ is true or not, instead of investigating all mental state pairs belonging to the transition. The proof involves a straightforward induction on the structure of ψ, and relies on Lemmas 1,3. Basically, all the hard work has already been done in these lemmas' proofs.

Lemma 5. *Let Ψ be a set of mental state conditions.*
Additionally, let $\boldsymbol{\tau}(\rho)$ be a transition class, and let $\tau \in \boldsymbol{\tau}(\rho)$.
Additionally, let $\langle \mu, \mu' \rangle \in \tau$ and $\psi \in \Psi$ such that $\mu \not\models_{\Psi} \psi$ and $\mu' \models_{\Psi} \psi$.
Then:

$$Der^{+}(\boldsymbol{\tau}(\rho), \psi)$$

Proof. Let $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle \langle \mathsf{ApproxAdd}_{\Sigma}, \mathsf{ApproxAdd}_{\Gamma} \rangle, \langle \mathsf{ApproxDel}_{\Sigma}, \mathsf{ApproxAdd}_{\Gamma} \rangle \rangle$.
Additionally, let $\mu = \langle K, \Sigma, \Gamma \rangle$, and let $\mu' = \langle K, \Sigma', \Gamma' \rangle$.
We proceed with induction on the structure of ψ.

Base ψ is a mental atom.
We proceed with case distinction.

- Suppose $\psi = \mathbf{bel}(\chi)$ such that $K \cup \Sigma \not\models \chi$ and $K \cup \Sigma' \models \chi$.
  Then, by Lemma 1, $der^{+}(K, \mathsf{ApproxAdd}_{\Sigma}, \mathsf{ApproxDel}_{\Sigma}, \chi)$.
  Then, by definition of Der^{+} in Def. 18, $Der^{+}(\boldsymbol{\tau}(\rho), \mathbf{bel}(\chi))$.
- Suppose $\psi = \mathbf{goal}(\chi)$ such that $K \cup \gamma \not\models \chi$ for all $\gamma \in \Gamma$, and there exists a $\gamma' \in \Gamma'$ such that $K \cup \gamma' \models \chi$.
  Then, by Lemma 3, there exists a goal $\gamma'' \in \mathsf{ApproxAdd}_{\Gamma}$ such that $der^{+}(K, \gamma'', \emptyset, \chi)$.
  Then, by definition of Der^{+} in Def. 18, $Der^{+}(\boldsymbol{\tau}(\rho), \mathbf{goal}(\chi))$.

Thus, $Der^{+}(\boldsymbol{\tau}(\rho), \psi)$ when ψ is a mental atom.

Hypothesis Let IH be the hypothesis that the lemma is true for all sub-formulas of ψ.

Step ψ is not a mental atom.
We proceed with case distinction.

- Suppose $\psi = \neg\psi'$ such that $\mu \models_{\Psi} \psi'$ and $\mu' \not\models_{\Psi} \psi'$.
  Then, by Lemma 6, $Der^{-}(\boldsymbol{\tau}(\rho), \psi')$.[28]
  Then, by definition of Der^{+} in Def. 18, $Der^{+}(\boldsymbol{\tau}(\rho), \neg\psi')$.
- Suppose $\psi = \psi_1 \wedge \psi_2$ such that $\mu \not\models_{\Psi} \psi_1$ and $\mu \models_{\Psi} \psi_1$, or $\mu \not\models_{\Psi} \psi_2$ and $\mu \models_{\Psi} \psi_2$.
  Then, by IH, $Der^{+}(\boldsymbol{\tau}(\rho), \psi_1)$ or $Der^{+}(\boldsymbol{\tau}(\rho), \psi_2)$.
  Then, by definition of Der^{+} in Def. 18, $Der^{+}(\boldsymbol{\tau}(\rho), \psi_1 \wedge \psi_2)$.

This establishes the lemma. □

The second lemma of this subsection, i.e. Lemma 6, is very similar to Lemma 5. It states that if a mental state condition ψ becomes false by execution of a transition in a transition class $\boldsymbol{\tau}(\rho)$, then $Der^{-}(\boldsymbol{\tau}(\rho), \psi)$ is true. As a consequence, to establish whether ψ might become false by execution of a transition in $\boldsymbol{\tau}(\rho)$, we may determine whether $Der^{-}(\boldsymbol{\tau}(\rho), \psi)$ is true or not, instead of investigating all mental state pairs belonging to the transition. The proof is similar to that of Lemma 6.

Lemma 6. *Let Ψ be a set of mental state conditions.*
Additionally, let $\boldsymbol{\tau}(\rho)$ be a transition class, and let $\tau \in \boldsymbol{\tau}(\rho)$.
Additionally, let $\langle \mu, \mu' \rangle \in \tau$ and $\psi \in \Psi$ such that $\mu \models_{\Psi} \psi$ and $\mu' \not\models_{\Psi} \psi$.
Then:

[28] We refer here to Lemma 6. However, in the proof of that lemma, we refer to *this* lemma, which might seem to yield a circular proof. This is not the case. The inductive applications of the two proofs eventually "halt" because the inner structure of mental state conditions is finite (note that we apply Lemma 6 on the argument of an MSC).

$$Der^{-}(\boldsymbol{\tau}(\rho), \psi)$$

Proof. Let $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_{\Sigma}, \mathsf{ApproxAdd}_{\Gamma}\rangle, \langle \mathsf{ApproxDel}_{\Sigma}, \mathsf{ApproxAdd}_{\Gamma}\rangle\rangle$.
Additionally, let $\mu = \langle K, \Sigma, \Gamma\rangle$, and let $\mu' = \langle K, \Sigma', \Gamma'\rangle$.
We proceed with induction on the structure of ψ.

Base ψ is a mental atom.
We proceed with case distinction.

- Suppose $\psi = \mathbf{bel}(\chi)$ such that $K \cup \Sigma \models \chi$ and $K \cup \Sigma' \not\models \chi$.
  Then, by Lemma 2, $der^{-}(K, \mathsf{ApproxAdd}_{\Sigma}, \mathsf{ApproxDel}_{\Sigma}, \chi)$.
  Then, by definition of Der^{-} in Def. 19, $Der^{-}(\boldsymbol{\tau}(\rho), \mathbf{bel}(\chi))$.
- Suppose $\psi = \mathbf{goal}(\chi)$ such that there exists a $\gamma \in \Gamma$ such that $K \cup \gamma \models \chi$, and $K \cup \gamma' \not\models \chi$ for all $\gamma' \in \Gamma'$.
  Then, by Lemma 4, there exists a goal $\gamma'' \in \mathsf{ApproxDel}_{\Gamma}$ such that $der^{+}(K, \gamma'', \emptyset, \chi)$.
  Then, by definition of Der^{-} in Def. 19, $Der^{-}(\boldsymbol{\tau}(\rho), \mathbf{goal}(\chi))$.

Thus, $Der^{-}(\boldsymbol{\tau}(\rho), \psi)$ when ψ is a mental atom.

Hypothesis Let Ih be the hypothesis that the lemma is true for all sub-formulas of ψ.

Step ψ is not a mental atom.
We proceed with case distinction.

- Suppose $\psi = \neg\psi'$ such that $\mu \not\models_{\Psi} \psi'$ and $\mu' \models_{\Psi} \psi'$.
  Then, by Lemma 5, $Der^{+}(\boldsymbol{\tau}(\rho), \psi')$.[29]
  Then, by definition of Der^{-} in Def. 19, $Der^{-}(\boldsymbol{\tau}(\rho), \neg\psi')$.
- Suppose $\psi = \psi_1 \wedge \psi_2$ such that $\mu \models_{\Psi} \psi_1$ and $\mu \not\models_{\Psi} \psi_1$, or $\mu \models_{\Psi} \psi_2$ and $\mu \not\models_{\Psi} \psi_2$.
  Then, by Ih, $Der^{-}(\boldsymbol{\tau}(\rho), \psi_1)$ or $Der^{-}(\boldsymbol{\tau}(\rho), \psi_2)$.
  Then, by definition of Der^{-} in Def. 19, $Der^{-}(\boldsymbol{\tau}(\rho), \psi_1 \wedge \psi_2)$.

This establishes the lemma. □

C.2 Independence Heuristics

In this section, we prove lemmas that are used to prove correctness of the independence heuristics for Goal that we introduced in Sect. 5.4.3. Specifically, the lemmas introduced below are used in the proofs of Theorems 6,8, Recall that an independence heuristic is correct if all pairs in the independence relation that it yields satisfy the independence conditions.

C.2.1 $\mathrm{H}_{\mathsf{Indep}}$ Lemmas

We start with two lemmas that concern $\mathrm{H}_{\mathsf{Indep}}$. Lemma 7 states that if two transitions satisfy $\mathrm{H}^{en}_{\mathsf{Indep}}$, then they satisfy the enabledness condition. Next, Lemma 8 states that if two transitions satisfy $\mathrm{H}^{com}_{\mathsf{Indep}}$, then they satisfy the commutativity condition. Both lemmas involve a quite straightforward proof by contradiction, and are used in the proof of Theorem 6 in Sect. 5.4.3.

Lemma 7. *Let τ, τ' be transitions that satisfy $\mathrm{H}^{en}_{\mathsf{Indep}}$.*
Then:

$$\tau, \tau' \text{ satisfy the enabledness condition}$$

[29] We refer here to Lemma 5. However, in the proof of that lemma, we refer to *this* lemma, which might seem to yield a circular proof. This is not the case. The inductive applications of the two proofs eventually "halt" because the inner structure of mental state conditions is finite (note that we apply Lemma 5 on the argument of an Msc).

Proof. Suppose τ and τ' violate the enabledness condition.
Then, by definition of the enabledness condition in Sect. 5.4.3, there exists a mental state $\mu \in \Omega_M$ such that $\tau, \tau' \in \mathsf{En}(\mu)$, and $\tau \notin \mathsf{En}(\tau'(\mu))$.
Then, by definition of Read in Def. 8, execution of τ' in μ changes the truth value of a mental state condition $\psi \in \mathsf{Read}(\tau)$, i.e. $\mu \models_\Psi \psi$ and $\tau'(\mu) \not\models_\Psi \psi$.
Then, by definition of Write^- in Def. 9, $\psi \in \mathsf{Write}^-(\tau', \mathsf{Read}(\tau))$, i.e. $\mathsf{Write}^-(\tau', \mathsf{Read}(\tau)) \neq \emptyset$.
Then, by definition of $\mathrm{H}^{en}_{\mathsf{Indep}}$, the τ, τ' violate $\mathrm{H}^{en}_{\mathsf{Indep}}$, which yields a contradiction.
This establishes the lemma. □

Lemma 8. *Let τ, τ' be transitions that satisfy $\mathrm{H}^{com}_{\mathsf{Indep}}$.*
Then:

$$\tau, \tau' \text{ satisfy the commutativity condition}$$

Proof. Let $\Delta(t) = \langle\langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma\rangle, \langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma\rangle\rangle$ and $\Delta(t') = \langle\langle \mathsf{Add}'_\Sigma, \mathsf{Add}'_\Gamma\rangle, \langle \mathsf{Del}'_\Sigma, \mathsf{Del}'_\Gamma\rangle\rangle$ for all $t \in \tau$ and $t' \in \tau'$.
Suppose τ and τ' violate the commutativity condition.
Then, by definition of the commutativity condition in Sect. 5.4.3, there exists a mental state $\mu \in \Omega_M$ such that $\tau, \tau' \in \mathsf{En}(\mu)$, and $\tau(\tau'(\mu)) \neq \tau'(\tau(\mu))$.
We proceed with case distinction.

- Suppose τ adds a fact χ to the belief base, whereas τ' deletes the same χ.
  Then, by definition of Δ in Def. 5, $\chi \in \mathsf{Add}_\Sigma$ and $\chi \in \mathsf{Del}'_\Sigma$ such that $\mathsf{Add}_\Sigma \cap \mathsf{Del}'_\Sigma \neq \emptyset$.
  Then, by definition of $\mathrm{H}^{com}_{\mathsf{Indep}}$, the τ, τ' violate $\mathrm{H}^{com}_{\mathsf{Indep}}$, which yields a contradiction.

- Suppose τ deletes a fact χ from the belief base, whereas τ' adds the same χ.
  Then, by definition of Δ in Def. 5, $\chi \in \mathsf{Del}_\Sigma$ and $\chi \in \mathsf{Add}'_\Sigma$ such that $\mathsf{Del}_\Sigma \cap \mathsf{Add}'_\Sigma \neq \emptyset$.
  Then, by definition of $\mathrm{H}^{com}_{\mathsf{Indep}}$, the τ, τ' violate $\mathrm{H}^{com}_{\mathsf{Indep}}$, which yields a contradiction.

- Suppose τ adds a conjunction of facts γ to the goal base, whereas τ' deletes the same γ.
  Then, by definition of Δ in Def. 5, $\gamma \in \mathsf{Add}_\Gamma$ and $\gamma \in \mathsf{Del}'_\Gamma$ such that $\mathsf{Add}_\Gamma \cap \mathsf{Del}'_\Gamma \neq \emptyset$.
  Then, by definition of $\mathrm{H}^{com}_{\mathsf{Indep}}$, the τ, τ' violate $\mathrm{H}^{com}_{\mathsf{Indep}}$, which yields a contradiction.

- Suppose τ deletes a conjunction of facts γ from the goal base, whereas τ' adds the same γ.
  Then, by definition of Δ in Def. 5, $\gamma \in \mathsf{Del}_\Gamma$ and $\gamma \in \mathsf{Add}'_\Gamma$ such that $\mathsf{Del}_\Gamma \cap \mathsf{Add}'_\Gamma \neq \emptyset$.
  Then, by definition of $\mathrm{H}^{com}_{\mathsf{Indep}}$, the τ, τ' violate $\mathrm{H}^{com}_{\mathsf{Indep}}$, which yields a contradiction.

This establishes the lemma. □

C.2.2 $\mathrm{H}_{\mathsf{ApproxIndep}}$

We proceed with two lemmas that concern $\mathrm{H}_{\mathsf{ApproxIndep}}$. Lemma 9 states that if two transition classes satisfy $\mathrm{H}^{en}_{\mathsf{ApproxIndep}}$, then the transitions belonging to those transition classes satisfy $\mathrm{H}^{en}_{\mathsf{Indep}}$. Note that we do not prove directly that these transitions satisfy the enabledness condition. Similarly, Lemma 10 states that if two transition classes satisfy $\mathrm{H}^{com}_{\mathsf{ApproxIndep}}$, then the transitions belonging to those transition classes satisfy $\mathrm{H}^{com}_{\mathsf{Indep}}$. Both lemmas involve a quite straightforward proof by contradiction, and are used in the proof of Theorem 8 in Sect. 5.4.3. Recall from Sect. 2.3.1 that for notational convenience, we regard a conjunction of facts (e.g. a goal or goal template) as a set containing all conjuncts (and nothing else).

Lemma 9. *Let $\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')$ be transition classes that satisfy $\mathrm{H}^{en}_{\mathsf{ApproxIndep}}$*
Additionally, let $\tau \in \boldsymbol{\tau}(\rho)$, and let $\tau \in \boldsymbol{\tau}(\rho')$.
Then:

$$\tau, \tau' \text{ satisfy } \mathrm{H}^{en}_{\mathsf{Indep}}$$

Proof. Suppose τ and τ' violate $\mathrm{H}^{en}_{\mathsf{Indep}}$.
Then, by definition of $\mathrm{H}^{en}_{\mathsf{Indep}}$, $\mathsf{Write}^{-}(\tau', \mathsf{Read}(\tau)) \neq \emptyset$.
Then, because $\mathsf{Read}(\tau) = \mathsf{ApproxRead}(\boldsymbol{\tau}(\rho))$ by Def. 21, $\mathsf{Write}^{-}(\tau', \mathsf{ApproxRead}(\boldsymbol{\tau}(\rho))) \neq \emptyset$.
Then, by Theorem 2, $\mathsf{ApproxWrite}^{-}(\boldsymbol{\tau}(\rho'), \mathsf{ApproxRead}(\boldsymbol{\tau}(\rho))) \neq \emptyset$.
Hence, $\mathrm{H}^{en}_{\mathsf{ApproxIndep}}$ is violated, which yields a contradiction. □

Lemma 10. *Let $\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho')$ be transition classes that satisfy $\mathrm{H}^{com}_{\mathsf{ApproxIndep}}$.*
Additionally, let $\tau \in \boldsymbol{\tau}(\rho)$, and let $\tau \in \boldsymbol{\tau}(\rho')$.
Then:

$$\tau, \tau' \text{ satisfy } \mathrm{H}^{com}_{\mathsf{ApproxIndep}}$$

Proof. Let $\Delta(t) = \langle\langle \mathsf{Add}_\Sigma, \mathsf{Add}_\Gamma\rangle, \langle \mathsf{Del}_\Sigma, \mathsf{Del}_\Gamma\rangle\rangle$ and $\Delta(t') = \langle\langle \mathsf{Add}'_\Sigma, \mathsf{Add}'_\Gamma\rangle, \langle \mathsf{Del}'_\Sigma, \mathsf{Del}'_\Gamma\rangle\rangle$ for all $t \in \tau$ and $t' \in \tau'$.
Additionally, let $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho)) = \langle\langle \mathsf{ApproxAdd}_\Sigma, \mathsf{ApproxAdd}_\Gamma\rangle, \langle \mathsf{ApproxDel}_\Sigma, \mathsf{ApproxDel}_\Gamma\rangle\rangle$, and let $\mathsf{Approx}\Delta(\boldsymbol{\tau}(\rho')) = \langle\langle \mathsf{ApproxAdd}'_\Sigma, \mathsf{ApproxAdd}'_\Gamma\rangle, \langle \mathsf{ApproxDel}'_\Sigma, \mathsf{ApproxDel}'_\Gamma\rangle\rangle$.
Suppose τ and τ' violate $\mathrm{H}^{com}_{\mathsf{Indep}}$.
Then, by definition of $\mathrm{H}^{com}_{\mathsf{Indep}}$, $\mathsf{Add}_\Sigma \cap \mathsf{Del}'_\Sigma \neq \emptyset$ or $\mathsf{Del}_\Sigma \cap \mathsf{Add}'_\Sigma \neq \emptyset$ or $\mathsf{Add}_\Gamma \cap \mathsf{Del}'_\Gamma \neq \emptyset$ or $\mathsf{Del}_\Gamma \cap \mathsf{Add}'_\Gamma \neq \emptyset$.
We proceed with case distinction.

- Suppose $\mathsf{Add}_\Sigma \cap \mathsf{Del}'_\Sigma \neq \emptyset$, i.e. there exists a ground fact χ'' such that $\chi'' \in \mathsf{Add}_\Sigma$ and $\chi'' \in \mathsf{Del}'_\Sigma$, i.e. $\mathsf{Add}_\Sigma \models \chi''$ and $\mathsf{Del}'_\Sigma \models \chi''$.
  Then, by Props. 1,2, $\mathsf{ApproxAdd}_\Sigma \models \chi''$ and $\mathsf{ApproxDel}'_\Sigma \models \chi''$, i.e. there exist substitutions θ, θ' such that there exist a $\chi \in \mathsf{ApproxAdd}_\Sigma$ and $\chi' \in \mathsf{ApproxDel}'_\Sigma$ such that $\chi'' = \chi\theta$ and $\chi'' = \chi'\theta$.
  Then, because χ'' is ground, there exists a substitution θ'' such that $\chi\theta'' = \chi'\theta''$.
  Then, because $\chi \in \mathsf{ApproxAdd}_\Sigma$ and $\chi' \in \mathsf{ApproxDel}'_\Sigma$, $\mathrm{H}^{com}_{\mathsf{ApproxIndep}}$ is violated, which yields a contradiction.

- Suppose $\mathsf{Del}_\Sigma \cap \mathsf{Add}'_\Sigma \neq \emptyset$, i.e. there exists a ground fact χ'' such that $\chi'' \in \mathsf{Del}_\Sigma$ and $\chi'' \in \mathsf{Add}'_\Sigma$, i.e. $\mathsf{Del}_\Sigma \models \chi''$ and $\mathsf{Add}'_\Sigma \models \chi''$.
  Then, by Props. 1,2, $\mathsf{ApproxDel}_\Sigma \models \chi''$ and $\mathsf{ApproxAdd}'_\Sigma \models \chi''$, i.e. there exist substitutions θ, θ' such that there exist a $\chi \in \mathsf{ApproxDel}_\Sigma$ and $\chi' \in \mathsf{ApproxAdd}'_\Sigma$ such that $\chi'' = \chi\theta$ and $\chi'' = \chi'\theta$.
  Then, because χ'' is ground, there exists a substitution θ'' such that $\chi\theta'' = \chi'\theta''$.
  Then, because $\chi \in \mathsf{ApproxDel}_\Sigma$ and $\chi' \in \mathsf{ApproxAdd}'_\Sigma$, $\mathrm{H}^{com}_{\mathsf{ApproxIndep}}$ is violated, which yields a contradiction.

- Suppose $\mathsf{Add}_\Gamma \cap \mathsf{Del}'_\Gamma \neq \emptyset$, i.e. there exists set of ground facts γ'' (i.e. an instantiated goal) such that $\gamma'' \in \mathsf{Add}_\Gamma$ and $\gamma'' \in \mathsf{Del}'_\Gamma$.
  Then, there exists a ground fact $\chi'' \in \gamma''$ such that $\gamma'' \models \chi''$.
  Then, by Props. 3,4, there exist a $\gamma \in \mathsf{ApproxAdd}_\Gamma$ and $\gamma' \in \mathsf{ApproxDel}'_\Gamma$ such that $\gamma \models \chi''$ and $\gamma' \models \chi''$, i.e. there exist substitutions θ, θ' such that there exist a $\chi \in \mathsf{ApproxAdd}_\Gamma$ and $\chi' \in \mathsf{ApproxDel}'_\Gamma$ such that $\chi'' = \chi\theta$ and $\chi'' = \chi'\theta'$.
  Then, because χ'' is ground, there exists a substitution θ'' such that $\chi\theta'' = \chi'\theta''$.
  Then, because $\chi \in \gamma$ and $\gamma \in \mathsf{ApproxAdd}_\Gamma$ and $\chi' \in \gamma'$ and $\gamma' \in \mathsf{ApproxDel}'_\Gamma$, $\mathrm{H}^{com}_{\mathsf{ApproxIndep}}$ is violated, which yields a contradiction.

- Suppose $\mathsf{Del}_\Gamma \cap \mathsf{Add}'_\Gamma \neq \emptyset$, i.e. there exists set of ground facts γ'' (i.e. an instantiated goal) such that $\gamma'' \in \mathsf{Del}_\Gamma$ and $\gamma'' \in \mathsf{Add}'_\Gamma$.
  Then, there exists a ground fact $\chi'' \in \gamma''$ such that $\gamma'' \models \chi''$.
  Then, by Props. 3,4, there exist a $\gamma \in \mathsf{ApproxDel}_\Gamma$ and $\gamma' \in \mathsf{ApproxAdd}'_\Gamma$ such that $\gamma \models \chi''$ and $\gamma' \models \chi''$, i.e. there exist substitutions θ, θ' such that there exist a $\chi \in \mathsf{ApproxDel}_\Gamma$ and $\chi' \in \mathsf{ApproxAdd}'_\Gamma$ such that $\chi'' = \chi\theta$ and $\chi'' = \chi'\theta'$.

Then, because χ'' is ground, there exists a substitution θ'' such that $\chi\theta'' = \chi'\theta''$.
Then, because $\chi \in \gamma$ and $\gamma \in \mathsf{ApproxDel}_\Gamma$ and $\chi' \in \gamma'$ and $\gamma' \in \mathsf{ApproxAdd}'_\Gamma$, $\mathrm{H}^{com}_{\mathsf{ApproxIndep}}$ is violated, which yields a contradiction.

This establishes the proposition. □

C.3 Slicing Algorithm

In this section, we introduce lemmas that are used to prove correctness of the PBS algorithm presented in Chap. 6. The section consists of three parts. First, in Sect. C.3.1, we prove lemmas that establish the correspondence between the influence graph and influence relation. These lemmas are used in the proof of Theorem 12 in Sect. 6.2, which is used to prove correctness of the PBS algorithm (Theorem 13 in Sect. 6.3). The second part of this section, i.e. Sect. C.3.2, establishes the existence of minimal ϕ-satisfying computation (see also Sect. 6.2.3). The main result is Lemma 13, which is later used to prove that our algorithm preserves logical soundness (see also Sect. 6.3). Finally, in Sect. C.3.3, we prove two more lemmas that concern the existence of corresponding computations in the original agent and its slice.

C.3.1 Correspondence $\mathcal{G}$ and Influential

We start with two lemmas concerning the correspondence between the influence relation Influence (defined in Def. 31) and the influence graph $\mathcal{G}$ (defined in Def. 30) with respect to an LTL formula ϕ. First, Lemma 11 states that if $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi$ is a route through the influence graph, then $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i)$ is true for some i. Subsequently, Lemma 12 states that if $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i)$ is true for some i, then there exists a route $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi$ through the influence graph. Both lemmas are proven by induction: the former on the the length of a route $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi$, and the latter on the value of i in $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i)$. The lemmas are used in the proof of Theorem 12 in Sect. 6.2, which summarises and extends the correspondence between the influence graph and influence relation to the influence graph and influential set.

Lemma 11. *Let ϕ be an LTL formula, let $\boldsymbol{\Omega_T}$ be the set of transition classes, and let $\mathcal{G}(\boldsymbol{\Omega_T}, \phi) = \langle \mathcal{V}, \mathcal{E} \rangle$ be the influence graph with respect to ϕ.*
Additionally, let $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi \in \mathit{Routes}(\mathcal{G}(\boldsymbol{\Omega_T}, \phi))$.
Then:

$$\text{there exists an } i \in \mathbb{N}^+ \text{ such that } \mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i)$$

Proof. By induction on the length $l = |\boldsymbol{\tau}(\rho) \rightsquigarrow \phi|$ of the route $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi$.

Base $l = 2$, i.e. $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi = \boldsymbol{\tau}(\rho)\phi$.
Hence, $\langle \boldsymbol{\tau}(\rho), \phi \rangle \in \mathcal{E}$.
Then, by definition of $\mathcal{E}$ in Def. 30, $\mathsf{ApproxVisible}(\boldsymbol{\tau}(\rho), \phi)$.
Then, by definition of Influence in Def. 31, $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, 1)$.
Thus, the lemma is true for $l = 2$.

Hypothesis Let IH be the hypothesis that the lemma is true for $l = k$.

Step $l = k + 1$, i.e. $\boldsymbol{\tau}(\rho) \rightsquigarrow \psi = \boldsymbol{\tau}(\rho)\boldsymbol{\tau}(\rho') \rightsquigarrow \psi$.
Hence, $\langle \boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho') \rangle \in \mathcal{E}$.
Then, by definition of $\mathcal{E}$ in Def. 30, $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho'), \boldsymbol{\tau}(\rho))$.
Also, by IH, there exists an $i \in \mathbb{N}^+$ such that $\mathsf{Influence}(\boldsymbol{\tau}(\rho'), \phi, i)$.
Then, by definition of Influence in Def. 30, $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i + 1)$.
Thus, IH is also true for $l = k + 1$.

This establishes the lemma. □

Lemma 12. *Let ϕ be an LTL formula, let $\mathbf{\Omega_T}$ be the set of transition classes, and let $\mathcal{G}(\mathbf{\Omega_T}, \phi) = \langle \mathcal{V}, \mathcal{E} \rangle$ be the influence graph with respect to ϕ.*
Additionally, let $i \in \mathbb{N}^+$ such that $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i)$. *Then:*

$$\boldsymbol{\tau}(\rho) \rightsquigarrow \phi \in \mathit{Routes}(\mathcal{G}(\mathbf{\Omega_T}, \phi))$$

Proof. By induction on i.

Base $i = 1$.
Then, by definition of $\mathsf{Influence}$ in Def. 31, $\mathsf{ApproxVisible}(\boldsymbol{\tau}(\rho), \phi)$.
Then, by definition of $\mathcal{E}$ in Def. 30, $\langle \boldsymbol{\tau}(\rho), \phi \rangle \in \mathcal{E}$.
Hence, there exists a route $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi \in \mathit{Routes}(\mathcal{G}(\mathbf{\Omega_T}, \phi))$, namely $\boldsymbol{\tau}(\rho)\phi$.
Thus, the lemma is true for $i = 1$.

Hypothesis Let Ih be the hypothesis that the lemma is true for $i = k$.

Step Suppose $i = k + 1$.
Then, by definition of $\mathsf{Influence}$ in Def. 31, there exists a $\boldsymbol{\tau}(\rho') \in \mathbf{\Omega_T}$ and $i > 1$ such that $\mathsf{Influence}(\boldsymbol{\tau}(\rho'), \phi, i - 1)$ and $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho'), \boldsymbol{\tau}(\rho))$.
Then, by definition of $\mathcal{E}$, there exists a $\langle \boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho') \rangle \in \mathcal{E}$.
Also, by Ih, there exists a route $\boldsymbol{\tau}(\rho') \rightsquigarrow \phi \in \mathit{Routes}(\mathcal{G}(\mathbf{\Omega_T}, \phi))$.
Hence, there exists a route $\boldsymbol{\tau}(\rho)\boldsymbol{\tau}(\rho') \rightsquigarrow \phi = \boldsymbol{\tau}(\rho) \rightsquigarrow \phi$ such that $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi \in \mathit{Routes}(\mathcal{G}(\mathbf{\Omega_T}, \phi))$.
Thus, Ih is also true for $i = k + 1$.

This establishes the lemma. □

C.3.2 Minimal ϕ-Satisfying Computation

In this section, we prove that if a Goal agent P has a computation on which an LTL\{X,R} formula ϕ is satisfied, then P also has a computation on which ϕ is satisfied on a finite prefix that is generated solely by transitions from $\mathsf{Influential}(\phi)$. Recall from Sect. 6.2.3 that this latter computation is called the minimal ϕ-satisfying computation. The existence of such a computation suggests that if we slice away all transitions that are not in $\mathsf{Influential}(\phi)$, sufficient transitions remain to establish ϕ's truth. In the proof of correctness of our algorithm, this is used as a building block.

We start with a proposition, i.e. Prop. 5, that states that a transition outside the influential set cannot enable a transition in the influential set. In our later lemma, i.e. Lemma 13, this result is used to establish that if there exists a computation $\boldsymbol{\pi}$ on which a mental state pair $\langle \boldsymbol{\pi}_i, \boldsymbol{\pi}_{i+1} \rangle$ occurs that belongs to a transition τ outside the influential set, i.e. $\langle \boldsymbol{\pi}_i, \boldsymbol{\pi}_{i+1} \rangle \in \tau$ and $\tau \notin \mathsf{Influential}(\phi)$, then there also exists a computation that is identical to $\boldsymbol{\pi}$ with the exception that τ is not executed in mental state $\boldsymbol{\pi}_i$ (recall that a computation is an infinite sequence of mental states, and that $\boldsymbol{\pi}_i$ is the i-th mental state on this sequence; see also Sect. 2.5). The proof of Prop. 5 is by contradiction, and is based on the idea that if a transition outside the influential set can enable a transition in the influential set, then this transition pair is in the enabled-by relation. Consequently, the transition outside the influential set would also be in the influential set, which yields a contradiction.

Proposition 5. *Let ϕ be an LTL formula, and let $\tau \notin \mathsf{Influential}(\phi)$ be a transition.*
Additionally, let $\langle \mu, \mu' \rangle \in \tau$.
Then:

$$\mathsf{En}(\mu') \cap \mathsf{Influential}(\phi) \quad \subseteq \quad \mathsf{En}(\mu) \cap \mathsf{Influential}(\phi)$$

Proof. Suppose the proposition is false, i.e. there exists a $\tau' \in \mathsf{En}(\mu') \cap \mathsf{Influential}(\phi)$ and $\tau' \notin \mathsf{En}(\mu) \cap \mathsf{Influential}(\phi)$.
By definition of $\mathsf{Influential}$ in Def. 32, there exist transition classes $\boldsymbol{\tau}(\rho) \not\subseteq \mathsf{Influential}(\phi)$ and

$\boldsymbol{\tau}(\rho') \subseteq \mathsf{Influential}(\phi)$ such that $\tau \in \boldsymbol{\tau}(\rho)$ and $\tau' \in \boldsymbol{\tau}(\rho')$.
Then, by definition of Read in Def. 8, there exists a $\psi \in \mathsf{Read}(\tau')$ such that $\mu \not\models_\Psi \psi$ and $\mu \models_\Psi \psi$.
Then, by definition of Write^+ in Def. 9, $\psi \in \mathsf{Write}^+(\tau, \mathsf{Read}(\tau'))$.
Then, by definition of $\mathsf{EnabledBy}$ in Def. 24, $\mathsf{EnabledBy}(\tau, \tau')$.
Then, Theorem 5, $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$.
Also, by definition of $\mathsf{Influential}$ in Def 32 and because $\boldsymbol{\tau}(\rho') \subseteq \mathsf{Influential}(\phi)$, there exists an $i \geq 1$ such that $\mathsf{Influence}(\boldsymbol{\tau}(\rho'), \phi, i)$.
Then, by definition of $\mathsf{Influence}$ in Def. 31, $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, i+1)$.
Then, by definition of $\mathsf{Influential}$ in Def 32, $\boldsymbol{\tau}(\rho) \subseteq \mathsf{Influential}(\phi)$, which yields a contradiction.
This establishes the proposition. □

The previous proposition is one property about transitions outside the influential set that we need in the proof of our theorem later. Another such property is that transitions outside the influential set do not *progress* the formula under investigation. Progression (e.g. [2]) formalises which LTL formula must be true in the next mental state on a computation for the formula that must be true in the current mental state to be true. We define progression as the function *progression* that maps an LTL\{X,R} formula ϕ, a natural number i (representing a point in time), and a computation $\boldsymbol{\pi}$ to the formula ϕ' that must be true in $\boldsymbol{\pi}_i$ for $\boldsymbol{\pi}_0$ to satisfy ϕ.

Definition 33. *Let ϕ be an LTL\{X,R} formula, let $\boldsymbol{\pi}$ be a computation, and let $i > 0$. Then:*

$$
\begin{aligned}
\mathit{progression}(\phi, 0, \boldsymbol{\pi}) &= \phi \\
\mathit{progression}(\phi, i, \boldsymbol{\pi}) &= \mathit{progress}(\mathit{progression}(\phi, i-1, \boldsymbol{\pi}), \boldsymbol{\pi}_{i-1})
\end{aligned}
$$

$$
\mathit{progress}(\phi, \mu) = \begin{cases}
\top & \phi \in \mathcal{L}_\Psi \text{ and } \mu \models_\Psi \phi \\
\bot & \phi \in \mathcal{L}_\Psi \text{ and } \mu \not\models_\Psi \phi \\
\neg \mathit{progress}(\phi', \mu) & \text{if } \phi = \neg\phi' \\
\mathit{progress}(\phi_1, \mu) \wedge \mathit{progress}(\phi_2, \mu) & \text{if } \phi = \phi_1 \wedge \phi_2 \\
\mathit{progress}(\phi_1, \mu) \vee \mathit{progress}(\phi_2, \mu) & \text{if } \phi = \phi_1 \vee \phi_2 \\
\mathit{progress}(\phi_2, \mu) \vee (\mathit{progress}(\phi_1, \mu) \wedge \phi) & \text{if } \phi = \phi_1 \,\mathsf{U}\, \phi_2
\end{cases}
$$

In the above definition, the function *progress* (whose definition is derived from [54]) computes the progression of an LTL\{X,R} formula in a mental state μ (we do not define *progress* for X and R because such formulas do not occur in LTL\{X,R}). Specifically, *progress* is used to define the progression at time point i by applying it on the progression at the previous time point (i.e. $i-1$), and the previous mental state (i.e. $\boldsymbol{\pi}_{i-1}$).

We proceed with a proposition, i.e. Prop. 6, which states that the execution of a transition τ that is outside the influential set in a mental state μ does not progress the formula that need be true in μ. That is, if ϕ is the formula that must be true in μ, then ϕ is also the formula that must be true in $\tau(\mu)$. We use this proposition in Lemma 13 to establish that if a transition τ that is outside the influential set is succeeded by a series of transitions that are in the influential set such that after execution of this series the progression is $\top$ (i.e. the formula's truth is established), then there also exists a computation on which this final series of transitions is executed without being preceded by τ.

The proof of Prop. 6 is by induction on the structure of the formula that must be true in μ. The base of the induction establishes this for U formulas whose arguments are non-temporal. The idea is that a $\phi_1 \,\mathsf{U}\, \phi_2$ formula is only progressed if ϕ_2 is true. In such case, because ϕ_2 is non-temporal, ϕ_2 must be visible to the executed transition, which is not the case if this transition is outside the influential set. In the inductive step, we prove that this is also the case if the arguments of a formula are non-temporal, given the hypothesis that the proposition is true for these arguments.

Proposition 6. *Let ϕ be an LTL\{X,R} formula, and let $\boldsymbol{\pi}$ be a computation such that $\boldsymbol{\pi} \models_{LTL} \phi$. Additionally, let $i \geq 0$ such that $\boldsymbol{\pi}_i = \mu$ and $\boldsymbol{\pi}_{i+1} = \mu'$.*

Additionally, let $\tau \notin \mathsf{Influential}(\phi)$ *be a transition such that* $\langle \mu, \mu' \rangle \in \tau$.
Additionally, let $\phi_{i+1} = progression(\phi, i+1, \boldsymbol{\pi})$ *and* $\phi_{i+1} \neq \top$.
Then:

$$\begin{array}{lrcl} & progression(\phi, i+1, \boldsymbol{\pi}) & = & progression(\phi, i+2, \boldsymbol{\pi}) \\ \text{i.e.} & progression(\phi, i+1, \boldsymbol{\pi}) & = & progress(progression(\phi, i+1, \boldsymbol{\pi}), \boldsymbol{\pi}_{i+1}) \\ \text{i.e.} & \phi_{i+1} & = & progress(\phi_{i+1}, \mu') \end{array}$$

Proof. By induction on the structure of ϕ_{i+1}.

Base $\phi_{i+1} = \phi' \,\mathsf{U}\, \phi''$ and ϕ', ϕ'' are non-temporal formulas.
Let $\boldsymbol{\tau}(\rho)$ be a transition class such that $\tau \in \boldsymbol{\tau}(\rho)$.
By definition of *progress* in Def. 33 and because $\phi_{i+1} = \phi' \,\mathsf{U}\, \phi''$, $\mu \not\models_{\text{LTL}} \phi''$.[30]
Suppose the proposition is false, i.e. $\phi_{i+1} \neq progress(\phi_{i+1}, \mu')$.
Then, by definition of *progress* in Def. 33, $progress(\phi_{i+1}, \mu') = \top$ and $\mu' \models_{\text{LTL}} \phi''$.[31]
Then, by definition of $\mathsf{Visible}$ in Def. 22 and because $\mu \not\models_{\text{LTL}} \phi''$ and because ϕ'' is non-temporal, $\mathsf{Visible}(\tau, \phi)$.
Then, because $\tau \in \boldsymbol{\tau}(\rho)$ and by Theorem 4, $\mathsf{ApproxVisible}(\boldsymbol{\tau}(\rho), \phi)$.
Then, by definition of $\mathsf{Influence}$ in Def. 31, $\mathsf{Influence}(\boldsymbol{\tau}(\rho), \phi, 1)$.
Then, by definition of $\mathsf{Influential}$ in Def. 32, $\boldsymbol{\tau}(\rho) \subseteq \mathsf{Influential}(\phi)$.
Hence, $\tau \in \mathsf{Influential}(\phi)$, which yields a contradiction.
Thus, the proposition is true if $\phi_{i+1} = \phi' \,\mathsf{U}\, \phi''$ and ϕ', ϕ'' are non-temporal formulas.

Hypothesis Let Ih be the hypothesis that the proposition is true for all sub-formulas ϕ', ϕ'' of ϕ_{i+1}.

Step $\phi_{i+1} \in \{\phi' \wedge \phi'', \phi' \vee \phi'', \phi' \,\mathsf{U}\, \phi''\}$.
By Ih, $\phi' = progress(\phi', \mu')$ and $\phi'' = progress(\phi'', \mu')$.
We proceed with case distinction.

- Suppose $\phi_{i+1} = \phi' \wedge \phi''$.
  Then, by definition of *progress* in Def. 33:

$$\begin{array}{rcl} progress(\phi_{i+1}, \mu') & = & progress(\phi' \wedge \phi'', \mu') \\ & = & progress(\phi', \mu') \wedge progress(\phi'', \mu') \\ & = & \phi' \wedge \phi'' \\ & = & \phi_{i+1} \end{array}$$

- Suppose $\phi_{i+1} = \phi' \vee \phi''$.
  Then, by definition of *progress* in Def. 33:

$$\begin{array}{rcl} progress(\phi_{i+1}, \mu') & = & progress(\phi' \vee \phi'', \mu') \\ & = & progress(\phi', \mu') \vee progress(\phi'', \mu') \\ & = & \phi' \vee \phi'' \\ & = & \phi_{i+1} \end{array}$$

- Suppose $\phi_{i+1} = \phi' \,\mathsf{U}\, \phi''$.
  Then, by definition of *progress* in Def. 33:

$$\begin{array}{rcl} progress(\phi_{i+1}, \mu') & = & progress(\phi' \,\mathsf{U}\, \phi'', \mu') \\ & = & progress(\phi'', \mu') \vee (progress(\phi', \mu') \wedge \phi_{i+1}) \\ & = & \phi'' \vee (\phi' \wedge \phi_{i+1}) \\ & = & \phi' \,\mathsf{U}\, \phi'' \\ & = & \phi_{i+1} \end{array}$$

[30] Otherwise, $\phi_{i+1} = \top$, which it is not by the proposition's premise.
[31] Applying *progress* to a U formula results in either $\top$, $\bot$, or the same U formula. Because by $\boldsymbol{\pi} \models_{\text{LTL}} \phi$ by the proposition's premise, $progress(\phi_{i+1}, \mu') = \bot$ cannot happen in this case. Thus, for $progress(\phi_{i+1}, \mu')$ to be unequal to $\phi_{i+1} = \phi' \,\mathsf{U}\, \phi''$ it must be $\top$.

Thus, IH is also true for $\phi_{i+1} \in \{\phi' \wedge \phi'', \phi' \vee \phi'', \phi' \,\mathsf{U}\, \phi''\}$.

This establishes the proposition. □

With Props. 5,6 we can now prove the following lemma, i.e. Lemma 13, which establishes the existence of a minimal ϕ-satisfying computation if at least one computation, say $\boldsymbol{\pi}$, exists on which the LTL\{X,R} formula ϕ is satisfied. We use this lemma in the proof of Theorem 13 in Sect. 6.3, which establishes that the PBS algorithm is correct. The proof of Lemma 13 is by induction on the number of transitions l outside the influential set that are used to generate a prefix of $\boldsymbol{\pi}$ on which the truth of ϕ can be established. The base of the induction is that no such transition exist, i.e. $l = 0$. In such case, $\boldsymbol{\pi}$ itself is a minimal ϕ-establishing computation such that the lemma immediately follows. The induction hypothesis is that the theorem is true for $l = k$, and in the inductive step we prove that the theorem is also true for $l = k + 1$. The idea is that by Props. 5,6, there exists a computation, say $\boldsymbol{\pi}''$, that is identical to $\boldsymbol{\pi}$ except that the *last* transition outside the influential set that is used to generate the prefix is not executed. Then, the induction hypothesis is applicable to $\boldsymbol{\pi}''$.

Lemma 13. *Let ϕ be an LTL\{X,R} formula, and let $\boldsymbol{\Pi}$ be the set of computations of a* GOAL *agent such that there exists a $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ such that $\boldsymbol{\pi} \models_{LTL} \phi$.*
Additionally, let $n \geq 0$ be the smallest number such that $progression(\phi, n, \boldsymbol{\pi}) = \top$.
Then:

$$\begin{aligned}&\text{there exist a } \boldsymbol{\pi}' \in \boldsymbol{\Pi} \text{ and } n' \leq n \text{ such that } progression(\phi, n', \boldsymbol{\pi}') = \top \\ &\quad \text{and } \{\tau \in \Omega_T \mid \tau \notin \mathsf{Influential}(\phi) \text{ and } 0 \leq i < n' \text{ and } \langle \boldsymbol{\pi}'_i, \boldsymbol{\pi}'_{i+1} \rangle \in \tau\} = \emptyset\end{aligned}$$

Proof. Let $T = \{\tau \in \Omega_T \mid \tau \notin \mathsf{Influential}(\phi) \text{ and } 0 \leq i < n \text{ and } \langle \boldsymbol{\pi}_i, \boldsymbol{\pi}_{i+1} \rangle \in \tau\}$.
We proceed with induction on the size $l = |T|$ of T.

Base $l = 0$, i.e. $T = \emptyset$.
Trivially, there exists a $\boldsymbol{\pi}' \in \boldsymbol{\Pi}$ (namely $\boldsymbol{\pi}$ itself) and $n' \leq n$ (namely n) such that $progression(\phi, n', \boldsymbol{\pi}') = \top$ and $\{\tau \in \Omega_T \mid \tau \notin \mathsf{Influential}(\phi) \text{ and } 0 \leq i < n' \text{ and } \langle \boldsymbol{\pi}'_i, \boldsymbol{\pi}'_{i+1} \rangle \in \tau\} = \emptyset$.

Hypothesis Let IH be the hypothesis that there exist a $\boldsymbol{\pi}' \in \boldsymbol{\Pi}$ and $n' \leq n$ such that for $l = k$, $progression(\phi, n', \boldsymbol{\pi}') = \top$ and $\{\tau \in \Omega_T \mid \tau \notin \mathsf{Influential}(\phi) \text{ and } 0 \leq i < n' \text{ and } \langle \boldsymbol{\pi}'_i, \boldsymbol{\pi}'_{i+1} \rangle \in \tau\} = \emptyset$.

Step $l = k + 1$.
Let $\tau_0 \cdots \tau_{n-1}$ be the sequence of transitions such that $\langle \boldsymbol{\pi}_i, \boldsymbol{\pi}_{i+1} \rangle \in \tau_i$ for all $0 \leq i < n$.
Additionally, let j be the largest number such that $\tau_j \notin \mathsf{Influential}(\phi)$, i.e. $\tau_{j'} \in \mathsf{Influential}(\phi)$ for all $j < j' \leq n - 1$.[32]
By Prop. 5, τ_j did not influence enabledness of $\tau_{j'} \in \mathsf{Influential}(\phi)$ for all $j < j' \leq n - 1$.
Hence, there exists a sequence of transitions $\tau_0 \cdots \tau_{j-1}\tau_{j+1} \cdots \tau_{n-1}$ such that:[33]

- $\tau_0 \in \mathsf{En}(\mu_0)$
- $\tau_i \in \mathsf{En}(\tau_{i-1}(\tau_{i-2}(\ldots(\tau_0(\mu_0))\ldots)))$ for all $1 \leq i \leq j - 1$
- $\tau_{j+1} \in \mathsf{En}(\tau_{j-1}(\tau_{j-2}(\ldots(\tau_0(\mu_0))\ldots)))$
- $\tau_i \in \mathsf{En}(\tau_{i-1}(\ldots(\tau_{j+1}(\tau_{j-1}(\ldots(\tau_0(\mu_0))\ldots)))))$ for all $j + 2 \leq i \leq n - 1$

[32] That is, τ_j is the last transition on the sequence $\tau_0 \cdots \tau_{n-1}$ that is outside the influential set, and that is succeeded by a series of transitions that are in the influential set.
[33] Informally, this means the following. The sequence $\tau_0 \cdots \tau_{j-1}\tau_{j+1} \cdots \tau_{n-1}$ is identical to the sequence $\tau_0 \cdots \tau_{n-1}$ except that τ_j is left out. Because τ_j (which is outside the influential set) does not influence enabledness of any transition in the influential set, all transitions executed after τ_j (which are in the influential set) are also enabled if τ_j is not executed.

Hence, there exists a computation $\boldsymbol{\pi}''$ such that:[34]

- $\boldsymbol{\pi}''_0 = \mu_0$
- $\boldsymbol{\pi}''_i = \tau_{i-1}(\boldsymbol{\pi}''_{i-1}) = \tau_{i-1}(\tau_{i-2}(\dots(\tau_0(\mu_0))\dots))$ for all $1 \leq i \leq j$
- $\boldsymbol{\pi}''_{j+1} = \tau_{j+1}(\boldsymbol{\pi}''_j) = \tau_{j-1}(\tau_{j-2}(\dots(\tau_0(\mu_0))\dots))$
- $\boldsymbol{\pi}''_i = \tau''_i(\boldsymbol{\pi}''_{i-1}) = \tau_{i-1}(\dots(\tau_{j+1}(\tau_{j-1}(\dots(\tau_0(\mu_0))\dots))))$ for all $j+2 \leq i \leq n-1$

Also, by Prop. 6 and because $\tau_j \notin \mathsf{Influential}(\phi)$, neither did τ_j progress ϕ, nor is it visible to ϕ.
Then, by definition of $\boldsymbol{\pi}''$, $progression(\phi, i, \boldsymbol{\pi}) = progression(\phi, i-1, \boldsymbol{\pi}'')$ for all $j+2 \leq i \leq n$.[35]
Then, because $progression(\phi, n, \boldsymbol{\pi}) = \top$, $progression(\phi, n-1, \boldsymbol{\pi}'') = \top$.
Then, because $|\{\tau \in \Omega_T \mid \tau \notin \mathsf{Influential}(\phi) \text{ and } 0 \leq i < n-1 \text{ and } \langle \boldsymbol{\pi}''_i, \boldsymbol{\pi}''_{i+1}\rangle \in \tau\}| = k$ and by IH, there exists a $\boldsymbol{\pi}' \in \boldsymbol{\Pi}$ (namely $\boldsymbol{\pi}''$) and $n' \leq n$ (namely $n-1$) such that $progression(\phi, n', \boldsymbol{\pi}') = \top$ and $\{\tau \in \Omega_T \mid \tau \notin \mathsf{Influential}(\phi) \text{ and } 0 \leq i < n' \text{ and } \langle \boldsymbol{\pi}'_i, \boldsymbol{\pi}'_{i+1}\rangle \in \tau\} = \emptyset$.
Thus, IH is also true for $l = k+1$.

This establishes the lemma. □

C.3.3 Computations of P and $\mathsf{Slice}(P)$

Finally, in this subsection, we prove two lemmas that are required in the proof of correctness of the algorithm, and that concern the correspondence between computations of an agent P and its slice $\mathsf{Slice}(P)$. More specifically, we first establish that if a program P has a computation with a finite prefix that is generated by transitions in $\mathsf{Influential}$ only, then $\mathsf{Slice}(P)$ has a computation with an equal prefix (Lemma 14). Subsequently, in the opposite direction, we can prove a similar claim, namely that each computation of $\mathsf{Slice}(P)$ that is generated by transitions in $\mathsf{Influential}$ only, then P has a computation with an equal prefix (Lemma 15). Both lemmas are proven by induction on the length of the prefix. We denote a prefix of a computation $\boldsymbol{\pi}$ of length i by $prefix(\boldsymbol{\pi}, i)$.

Lemma 14. *Let ϕ be an LTL\{X,R} formula, let P be a GOAL agent, let $\mathsf{Slice}(P) = \overline{P}$ be computed by Alg. 3, let $\boldsymbol{\Pi}$ be the set of computations of P, and let $\overline{\boldsymbol{\Pi}}$ be the set of computations of $\overline{P}$.*
Additionally, let $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ and let $j \geq 0$ such that for all $0 \leq i \leq j$, there exists a $\tau \in \mathsf{Influential}(\phi)$ such that $\langle \boldsymbol{\pi}_i, \boldsymbol{\pi}_{i+1}\rangle \in \tau$.
Then:

$$\text{there exists a } \overline{\boldsymbol{\pi}} \in \overline{\boldsymbol{\Pi}} \text{ such that } prefix(\boldsymbol{\pi}, j) = prefix(\overline{\boldsymbol{\pi}}, j))$$

Proof. By induction on $prefix(\overline{\boldsymbol{\pi}}, i)$ with $i \leq j$.

Base: $i = 1$.
Let μ_0 be the initial state of the agent.
Then, because Slice does not alter μ_0, $prefix(\boldsymbol{\pi}, 1) = \boldsymbol{\pi}_0 = \mu_0$ and $prefix(\overline{\boldsymbol{\pi}}, 1) = \overline{\boldsymbol{\pi}}_0 = \mu_0$ for all $\overline{\boldsymbol{\pi}} \in \overline{\boldsymbol{\Pi}}$.
Thus, the lemma is true for $i = 1$.

Hypothesis Let IH be the hypothesis that there exists a $\overline{\boldsymbol{\pi}} \in \overline{\boldsymbol{\Pi}}$ such that $prefix(\boldsymbol{\pi}, i) = prefix(\overline{\boldsymbol{\pi}}, i)$ for $i = k$.

[34] Informally, this means the following. The prefix $\boldsymbol{\pi}''_0 \cdots \boldsymbol{\pi}''_j$ of $\boldsymbol{\pi}''$ is identical to the prefix $\boldsymbol{\pi}_0 \cdots \boldsymbol{\pi}_j$. Then, rather than executing τ_j in $\boldsymbol{\pi}''_j$ (which happens on $\boldsymbol{\pi}$ in $\boldsymbol{\pi}_j$), τ_{j+1} is executed instead, followed by the transitions that succeed τ_{j+1} on the sequence $\tau_0 \cdots \tau_{n-1}$. Thus, τ_j is "skipped".

[35] Although the mental states on the infix $\boldsymbol{\pi}''_{j+1} \cdots \boldsymbol{\pi}''_{n-1}$ of $\boldsymbol{\pi}''$ and the infix $\boldsymbol{\pi}_{j+2} \cdots \boldsymbol{\pi}_n$ are not identical, the valuation of these mental states with respect to the mental state conditions occurring in ϕ are (otherwise, τ_j would be visible to ϕ which it is not).

Step $i = k+1$.
Let τ_{k-1} be the transition executed in $\boldsymbol{\pi}_{k-1}$, i.e. $\langle \boldsymbol{\pi}_{k-1}, \boldsymbol{\pi}_k \rangle \in \tau_{k-1}$, and let $\boldsymbol{\tau}(\rho)$ be a transition class such that $\tau_{k-1} \in \boldsymbol{\tau}(\rho)$.
Then, because $\tau_{k-1} \in \mathsf{Influential}(\phi)$ by the lemma's premise, $\boldsymbol{\tau}(\rho) \subseteq \mathsf{Influential}(\phi)$.
Then, by Theorem 12, $\boldsymbol{\tau}(\rho) \rightsquigarrow \phi \in \mathit{Routes}(\mathcal{G}(\boldsymbol{\Omega}_{\boldsymbol{T}}, \phi))$.
Then, by definition of $\overline{P}$ in Alg. 3, $\rho \in \overline{R}$.
Also, by IH, there exists a $\overline{\boldsymbol{\pi}} \in \overline{\boldsymbol{\Pi}}$ such that $\mathit{prefix}(\boldsymbol{\pi}, k) = \mathit{prefix}(\overline{\boldsymbol{\pi}}, k)$.
Then, because $\boldsymbol{\pi}_{k-1} = \overline{\boldsymbol{\pi}}_{k-1}$ and $\tau_{k-1} \in \mathsf{En}(\boldsymbol{\pi}_{k-1})$ and $\rho \in \overline{R}$, $\tau_{k-1} \in \mathsf{En}(\overline{\boldsymbol{\pi}}_{k-1})$.
Then, by GOAL's transition rule in Sect. 2.5, there exists a $\overline{\boldsymbol{\pi}}' \in \overline{\boldsymbol{\Pi}}$ such that $\mathit{prefix}(\overline{\boldsymbol{\pi}}, k) = \mathit{prefix}(\overline{\boldsymbol{\pi}}', k)$ and τ_{k-1} is executed in $\overline{\boldsymbol{\pi}}'_{k-1}$, i.e. $\overline{\boldsymbol{\pi}}'_k = \tau_{k-1}(\overline{\boldsymbol{\pi}}'_{k-1})$.
Then, since $\boldsymbol{\pi}_{k-1} = \overline{\boldsymbol{\pi}}_{k-1} = \overline{\boldsymbol{\pi}}'_{k-1}$ and $\boldsymbol{\pi}_k = \tau_{k-1}(\boldsymbol{\pi}_{k-1})$, $\mathit{prefix}(\boldsymbol{\pi}, k{+}1) = \mathit{prefix}(\overline{\boldsymbol{\pi}}', k{+}1)$.
Thus, IH is also true for $i = k+1$.

This establishes the lemma. □

Lemma 15. *Let ϕ be an LTL\{X,R} formula, let P be a* GOAL *agent, let* $\mathsf{Slice}(P) = \overline{P}$ *be computed by Alg. 3, let $\boldsymbol{\Pi}$ be the set of computations of P, and let $\overline{\boldsymbol{\Pi}}$ be the set of computations of $\overline{P}$.*
Additionally, let $\overline{\boldsymbol{\pi}} \in \overline{\boldsymbol{\Pi}}$ and let $j \geq 0$ such that for all $0 \leq i \leq j$, there exists a $\tau \in \mathsf{Influential}(\phi)$ such that $\langle \boldsymbol{\pi}_i, \boldsymbol{\pi}_{i+1} \rangle \in \tau$.
Then:

$$\text{there exists a } \boldsymbol{\pi} \in \boldsymbol{\Pi} \text{ such that } \mathit{prefix}(\boldsymbol{\pi}, j) = \mathit{prefix}(\overline{\boldsymbol{\pi}}, j)$$

Proof. By induction on $\mathit{prefix}(\overline{\boldsymbol{\pi}}, i)$ with $i \leq j$.

Base: $i = 1$.
Let μ_0 be the initial state of the agent.
Then, because Slice does not alter μ_0, $\mathit{prefix}(\overline{\boldsymbol{\pi}}, 1) = \overline{\boldsymbol{\pi}}_0 = \mu_0$ and $\mathit{prefix}(\boldsymbol{\pi}, 1) = \boldsymbol{\pi}_0 = \mu_0$ for all $\boldsymbol{\pi} \in \boldsymbol{\Pi}$.
Thus, the lemma is true for $i = 1$.

Hypothesis Let IH be the hypothesis that there exists a $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ such that $\mathit{prefix}(\boldsymbol{\pi}, i) = \mathit{prefix}(\overline{\boldsymbol{\pi}}, i)$ for $i = k$.

Step $i = k+1$.
Let τ_{k-1} be the transition executed in $\overline{\boldsymbol{\pi}}_{k-1}$, i.e. $\langle \overline{\boldsymbol{\pi}}_{k-1}, \overline{\boldsymbol{\pi}}_k \rangle \in \tau_{k-1}$.
Then, by definition of $\overline{P}$ in Alg. 3 and because $\tau_{k-1} \in \mathsf{Influential}(\phi)$, $\mathsf{Rule}(\tau_{k-1}) \in R$.
Then, by IH, there exists a $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ such that $\mathit{prefix}(\boldsymbol{\pi}, k) = \mathit{prefix}(\overline{\boldsymbol{\pi}}, k)$.
Then, because $\boldsymbol{\pi}_{k-1} = \overline{\boldsymbol{\pi}}_{k-1}$ and $\tau_{k-1} \in \mathsf{En}(\overline{\boldsymbol{\pi}}_{k-1})$, $\tau_{k-1} \in \mathsf{En}(\boldsymbol{\pi}_{k-1})$.
Then, by GOAL's transition rule in Sect. 2.5, there exists a $\boldsymbol{\pi}' \in \boldsymbol{\Pi}$ such that $\mathit{prefix}(\boldsymbol{\pi}, k) = \mathit{prefix}(\boldsymbol{\pi}', k)$ and τ_{k-1} is executed in $\boldsymbol{\pi}'_{k-1}$, i.e. $\boldsymbol{\pi}'_k = \tau_{k-1}(\boldsymbol{\pi}'_{k-1})$.
Then, as $\boldsymbol{\pi}'_{k-1} = \boldsymbol{\pi}_{k-1} = \overline{\boldsymbol{\pi}}_{k-1}$ and $\overline{\boldsymbol{\pi}}_k = \tau_{k-1}(\overline{\boldsymbol{\pi}}_{k-1})$, $\mathit{prefix}(\boldsymbol{\pi}', k+1) = \mathit{prefix}(\overline{\boldsymbol{\pi}}, k+1)$.
Thus, IH is also true for $i = k+1$.

This establishes the lemma. □

C.4 Heuristics for C1

In this section, we introduce and prove two propositions that concern heuristics $\mathrm{H}^{sing}_{\mathrm{POR}}$ and $\mathrm{H}^{mult}_{\mathrm{POR}}$ (see Sect. 7.2.1). The first proposition, i.e. Prop. 8, establishes that a candidate set generated by $\mathrm{H}^{sing}_{\mathrm{POR}}$ satisfies **C1**; the second proposition, i.e. Prop. 10, establishes the same for candidate sets generated by $\mathrm{H}^{mult}_{\mathrm{POR}}$. The proofs follow the informal presentation of the heuristics in Sect. 7.2.1.

In support of Props. 8,10, we prove two additional propositions, namely Props. 7,9. Informally, Prop. 7 establishes that if a transition τ that is not enabled in a mental state μ, but is enabled after execution of a sequence of transitions *seq*, then this sequence contains a transition belonging

to the same transition class as τ that is enabled in μ, or *seq* contains a transition τ' belonging to another transition class than τ such that $\mathsf{EnabledBy}(\tau, \tau')$. Proposition 9 generalises this result to sets of transition classes. That is, it states the same as Prop. 7 except that we only know that τ belongs to some transition class in a set of transition classes without knowing exactly which transition class in this set.

C.4.1 $\mathrm{H}^{sing}_{\mathrm{POR}}$ Propositions

Proposition 7. *Let μ be a mental state, let $\boldsymbol{\Omega_T}$ be the set of all transition classes, and let $\boldsymbol{\tau}(\rho^*) \in \boldsymbol{\Omega_T}$ such that there exists a $\tau^* \in \boldsymbol{\tau}(\rho^*)$ such that $\tau^* \notin \mathsf{En}(\mu)$.*
Additionally, let $seq = \tau \cdots \tau^*$ be a sequence of transitions executed starting from μ, and let T' be the set of transitions that are executed before τ^*.
Additionally, let $\tau' \in T'$ such that $\mathsf{EnabledBy}(\tau^*, \tau')$.
Additionally, let $\boldsymbol{\tau}(\rho') \in \boldsymbol{\Omega_T}$ such that $\tau' \in \boldsymbol{\tau}(\rho')$.
Then:

$$\tau' \in \boldsymbol{\tau}(\rho^*) \cap \mathsf{En}(\mu) \quad \text{or} \quad \boldsymbol{\tau}(\rho') \neq \boldsymbol{\tau}(\rho^*) \text{ and } \mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho^*), \boldsymbol{\tau}(\rho'))$$

Proof. By complete induction on the size $l = |T'|$ of T'.

Base $l = 1$, i.e. $T' = \{\tau'\}$.
We proceed with case distinction on whether $\boldsymbol{\tau}(\rho') = \boldsymbol{\tau}(\rho^*)$.

- Suppose $\boldsymbol{\tau}(\rho') = \boldsymbol{\tau}(\rho^*)$.
  Because $\tau' \in \mathsf{En}(\mu)$, $\tau' \in \boldsymbol{\tau}(\rho^*) \cap \mathsf{En}(\mu)$ such that the base is true.
- Suppose $\boldsymbol{\tau}(\rho') \neq \boldsymbol{\tau}(\rho^*)$.
  Because $\mathsf{EnabledBy}(\tau^*, \tau')$ and by Theorem 5, $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho^*), \boldsymbol{\tau}(\rho'))$ such that the base is true.

Hypothesis Let Ih be the hypothesis that the lemma is true for $1 \leq l \leq k$.

Step $l = k + 1$.
We proceed with case distinction on whether $\boldsymbol{\tau}(\rho') = \boldsymbol{\tau}(\rho^*)$.

- Suppose $\boldsymbol{\tau}(\rho') = \boldsymbol{\tau}(\rho^*)$.
  We proceed with case distinction.
  - Suppose $\tau' \in \mathsf{En}(\mu)$.
    Then, $\tau' \in \boldsymbol{\tau}(\rho^*) \cap \mathsf{En}(\mu)$ such that Ih is also true for $l = k + 1$.
  - Suppose $\tau' \notin \mathsf{En}(\mu)$.
    Let seq_{pre} be the prefix of *seq* such that $seq_{pre} = \tau \cdots \tau'$, and let T'' be the set of transitions that are executed before τ'.
    Then, there exists a $\tau'' \in T''$ such that $\mathsf{EnabledBy}(\tau', \tau'')$.
    Then, by Ih and because $T'' \subset T'$, $\tau'' \in \boldsymbol{\tau}(\rho^*) \cap \mathsf{En}(\mu)$ or there exists a $\boldsymbol{\tau}(\rho'') \in \boldsymbol{\Omega_T}$ such that $\tau'' \in \boldsymbol{\tau}(\rho'')$ and $\boldsymbol{\tau}(\rho'') \neq \boldsymbol{\tau}(\rho^*)$ and $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho^*), \boldsymbol{\tau}(\rho''))$.
    Hence, Ih is also true for $l = k + 1$.
- Suppose $\boldsymbol{\tau}(\rho') \neq \boldsymbol{\tau}(\rho^*)$.
  Because $\mathsf{EnabledBy}(\tau^*, \tau)$ and by Theorem 5, $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho^*), \boldsymbol{\tau}(\rho'))$ such that Ih is also true for $k + 1$.

This establishes the proposition. □

Proposition 8. *Let $\boldsymbol{\Omega_T}$ be the set of all transition classes, and let μ be a mental state.*
Additionally, let $\boldsymbol{\tau}(\rho) \in \boldsymbol{\Omega_T}$ such that

- *for all $\boldsymbol{\tau}(\rho') \in \boldsymbol{\Omega_T} \setminus \{\boldsymbol{\tau}(\rho)\}$, neither $\mathsf{ApproxDep}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$ nor $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$.*

Then:

$$\boldsymbol{\tau}(\rho) \cap \mathsf{En}(\mu) \text{ satisfies } \mathbf{C1}$$

Proof. Suppose the lemma is false, i.e. $\boldsymbol{\tau}(\rho) \cap \mathsf{En}(\mu)$ violates **C1**.
Then, by definition of **C1**, in the full transition system, there exists a path π starting in μ whose generation involves the execution of a transition τ that is dependent on a transition in $\boldsymbol{\tau}(\rho) \cap \mathsf{En}(\mu)$ before the execution of a transition from $\boldsymbol{\tau}(\rho) \cap \mathsf{En}(\mu)$.
Specifically, let τ be the first such transition that was executed to generate π.
We proceed with case distinction on whether $\tau \in \boldsymbol{\tau}(\rho)$.

1. Suppose $\tau \in \boldsymbol{\tau}(\rho)$.
   We proceed with case distinction on whether $\tau \in \mathsf{En}(\mu)$.
   a) Suppose $\tau \in \mathsf{En}(\mu)$.
      Then, because $\tau \in \boldsymbol{\tau}(\rho)$, $\tau \in \boldsymbol{\tau}(\rho) \cap \mathsf{En}(\mu)$, which yields a contradiction.[36]
   b) Suppose $\tau \notin \mathsf{En}(\mu)$.
      Let T' be the set of transitions that are executed to generate π before τ.
      Then, because $\tau \notin \mathsf{En}(\mu)$, there exists a $\tau' \in T'$ such that $\mathsf{EnabledBy}(\tau, \tau')$.
      Then, by Prop. 7, $\tau' \in \boldsymbol{\tau}(\rho) \cap \mathsf{En}(\mu)$ or there exists a $\boldsymbol{\tau}(\rho') \in \boldsymbol{\Omega_T}$ such that $\tau' \in \boldsymbol{\tau}(\rho')$ and $\boldsymbol{\tau}(\rho) \neq \boldsymbol{\tau}(\rho')$ and $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$.
      We proceed with case distinction.
      i. Suppose $\tau' \in \boldsymbol{\tau}(\rho) \cap \mathsf{En}(\mu)$.
         Then, because τ' is executed before τ, τ cannot be the first transition executed during the generation of π that is dependent on a transition in $\boldsymbol{\tau}(\rho) \cap \mathsf{En}(\mu)$ before a transition in $\boldsymbol{\tau}(\rho) \cap \mathsf{En}(\mu)$ is executed, which yields a contradiction.
      ii. Suppose $\boldsymbol{\tau}(\rho) \neq \boldsymbol{\tau}(\rho')$ and $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$.
         Then, because $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho''))$ for all $\boldsymbol{\tau}(\rho'') \in \boldsymbol{\Omega_T} \setminus \{\boldsymbol{\tau}(\rho)\}$, a contradiction is yielded.
2. Suppose $\tau \notin \boldsymbol{\tau}(\rho)$.
   Then, because τ is dependent on a transition in $\boldsymbol{\tau}(\rho) \cap \mathsf{En}(\mu)$, there exists a $\tau' \in \boldsymbol{\tau}(\rho)$ such that $\mathsf{Dep}(\tau, \tau')$.
   Then, by Theorem 11, there exist a $\boldsymbol{\tau}(\rho') \in \boldsymbol{\Omega_T} \setminus \{\boldsymbol{\tau}(\rho)\}$ such that $\tau \in \boldsymbol{\tau}(\rho')$ and $\tau' \in \boldsymbol{\tau}(\rho)$ and $\mathsf{ApproxDep}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$, which yields a contradiction.

This establishes the proposition. □

C.4.2 $\mathrm{H}_{\mathrm{POR}}^{mult}$ Propositions

Proposition 9. *Let μ be a mental state, let $\boldsymbol{\Omega_T}$ be the set of all transition classes, and let $\boldsymbol{T} \subseteq \boldsymbol{\Omega_T}$ such that there exists a $\boldsymbol{\tau}(\rho^*) \in \boldsymbol{T}$ such that there exists a $\tau^* \in \boldsymbol{\tau}(\rho^*)$ such that $\tau^* \notin \mathsf{En}(\mu)$.*
Additionally, let $T = \bigcup_{\boldsymbol{\tau}(\rho) \in \boldsymbol{T}} \boldsymbol{\tau}(\rho)$, i.e. $\tau^* \in T$.
Additionally, let $seq = \tau \cdots \tau^*$ be a sequence of transitions executed starting from μ, and let T' be the set of transitions that are executed before τ^*.
Additionally, let $\tau' \in T'$ such that $\mathsf{EnabledBy}(\tau^*, \tau')$.
Additionally, let $\boldsymbol{\tau}(\rho') \in \boldsymbol{\Omega_T}$ such that $\tau \in \boldsymbol{\tau}(\rho')$.
Then:

$$\tau' \in T \cap \mathsf{En}(\mu) \quad \text{or} \quad \boldsymbol{\tau}(\rho') \notin \boldsymbol{T} \text{ and } \mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho^*), \boldsymbol{\tau}(\rho'))$$

Proof. By complete induction on the size $l = |T'|$ of T'.

[36] If $\tau \in \boldsymbol{\tau}(\rho) \cap \mathsf{En}(\mu)$, then it cannot be a transition that is executed *before* execution of a transition in $\boldsymbol{\tau}(\rho) \cap \mathsf{En}(\mu)$ as we assumed.

Base $l = 1$, i.e. $T' = \{\tau'\}$.
We proceed with case distinction on whether $\boldsymbol{\tau}(\rho') \in \boldsymbol{T}$.

1. Suppose $\boldsymbol{\tau}(\rho') \in \boldsymbol{T}$.
   Because $\tau' \in \mathsf{En}(\mu)$, $\tau' \in T \cap \mathsf{En}(\mu)$ such that the base is true.
2. Suppose $\boldsymbol{\tau}(\rho') \notin \boldsymbol{T}$.
   Because $\mathsf{EnabledBy}(\tau^*, \tau')$ and by Theorem 5, $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho^*), \boldsymbol{\tau}(\rho'))$ such that the base is true.

Hypothesis Let Ih be the hypothesis that the lemma is true for $1 \leq l \leq k$.

Step $l = k + 1$.
We proceed with case distinction on whether $\boldsymbol{\tau}(\rho') \in \boldsymbol{T}$.

1. Suppose $\boldsymbol{\tau}(\rho') \in \boldsymbol{T}$.
   We proceed with case distinction on whether $\tau' \in \mathsf{En}(\mu)$.
   - Suppose $\tau' \in \mathsf{En}(\mu)$.
     Then, because $\tau' \in T$, $\tau' \in T \cap \mathsf{En}(\mu)$ such that Ih is also true for $l = k + 1$.
   - Suppose $\tau' \notin \mathsf{En}(\mu)$.
     Let seq_{pre} be the prefix of seq such that $seq_{pre} = \tau \cdots \tau'$, and let T'' be the set of transitions that are executed before τ'.
     Then, there exists a $\tau'' \in T''$ such that $\mathsf{EnabledBy}(\tau', \tau'')$.
     Then, by Ih, $\tau'' \in T \cap \mathsf{En}(\mu)$ or there exists a $\boldsymbol{\tau}(\rho'') \in \boldsymbol{\Omega_T}$ such that $\tau'' \in \boldsymbol{\tau}(\rho'')$ and $\boldsymbol{\tau}(\rho'') \notin \boldsymbol{T}$ and $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho^*), \boldsymbol{\tau}(\rho''))$.
     Hence, Ih is also true for $l = k + 1$.
2. Suppose $\boldsymbol{\tau}(\rho') \notin \boldsymbol{T}$.
   Because $\mathsf{EnabledBy}(\tau^*, \tau')$ and by Theorem 5, $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho^*), \boldsymbol{\tau}(\rho'))$ such that Ih is also true for $k + 1$.

This establishes the proposition. □

Proposition 10. *Let $\boldsymbol{\Omega_T}$ be the set of all transition classes, and let μ be a mental state. Additionally, let $\boldsymbol{T} \subseteq \boldsymbol{\Omega_T}$ such that $T = \bigcup_{\boldsymbol{\tau}(\rho) \in \boldsymbol{T}} \boldsymbol{\tau}(\rho)$ and:*

- *for all $\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho') \in \boldsymbol{T}$, $\mathsf{ApproxDep}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$*
- *for all $\boldsymbol{\tau}(\rho') \in \boldsymbol{\Omega_T} \setminus \boldsymbol{T}$, there does not exist a $\boldsymbol{\tau}(\rho) \in \boldsymbol{T}$ such that $\mathsf{ApproxDep}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$ or $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$*

Then:

$$T \cap \mathsf{En}(\mu) \text{ satisfies } \mathbf{C1}$$

Proof. Suppose the lemma is false, i.e. $T \cap \mathsf{En}(\mu)$ violates **C1**.
Then, by definition of **C1**, in the full transition system, there exists a path π starting in μ whose generation involves the execution of a transition τ that is dependent on a transition in $T \cap \mathsf{En}(\mu)$ before the execution of a transition from $T \cap \mathsf{En}(\mu)$.
Specifically, let τ be the first such transition that was executed to generate π.
We proceed with case distinction on whether $\tau \in T$.

1. Suppose $\tau \in T$.
   Let $\boldsymbol{\tau}(\rho) \in \boldsymbol{T}$ such that $\tau \in \boldsymbol{\tau}(\rho)$.
   We proceed with case distinction on whether $\tau \in \mathsf{En}(\mu)$.

a) Suppose $\tau \in \mathsf{En}(\mu)$.
Then, because $\tau \in T$, $\tau \in T \cap \mathsf{En}(\mu)$, which yields a contradiction.[37]

b) Suppose $\tau \notin \mathsf{En}(\mu)$.
Let T' be the set of transitions that are executed to generate π before τ.
Then, because $\tau \notin \mathsf{En}(\mu)$, there exists a $\tau' \in T'$ such that $\mathsf{EnabledBy}(\tau, \tau')$.
Then, by Prop. 9, $\tau' \in T \cap \mathsf{En}(\mu)$ or there exists a $\boldsymbol{\tau}(\rho') \in \boldsymbol{\Omega_T}$ such that $\tau' \in \boldsymbol{\tau}(\rho')$ and $\boldsymbol{\tau}(\rho') \notin \boldsymbol{T}$ and $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$.
We proceed with case distinction.

i. Suppose $\tau' \in T \cap \mathsf{En}(\mu)$.
Then, because τ' is executed before τ, τ cannot be the first transition executed during the generation of π that is dependent on a transition in $T \cap \mathsf{En}(\mu)$ before a transition in $T \cap \mathsf{En}(\mu)$ is executed, which yields a contradiction.

ii. Suppose $\boldsymbol{\tau}(\rho') \notin \boldsymbol{T}$ and $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$.
Then, because $\mathsf{ApproxEnabledBy}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho''))$ for all $\boldsymbol{\tau}(\rho'') \in \boldsymbol{\Omega_T} \setminus \boldsymbol{T}$, a contradiction is yielded.

2. Suppose $\tau \notin T$.
Then, because τ is dependent on a transition in $T \cap \mathsf{En}(\mu)$, there exists a $\tau' \in T$ such that $\mathsf{Dep}(\tau, \tau')$.
Then, by Theorem 11, there exist a $\boldsymbol{\tau}(\rho) \in \boldsymbol{\Omega_T} \setminus \boldsymbol{T}$ and $\boldsymbol{\tau}(\rho') \in \boldsymbol{T}$ such that $\tau' \in \boldsymbol{\tau}(\rho')$ and $\tau \in \boldsymbol{\tau}(\rho)$ and $\mathsf{ApproxDep}(\boldsymbol{\tau}(\rho), \boldsymbol{\tau}(\rho'))$, which yields a contradiction.

This establishes the proposition. □

[37] If $\tau \in T \cap \mathsf{En}(\mu)$, then it cannot be a transition that is executed *before* execution of a transition in $T \cap \mathsf{En}(\mu)$ as we assumed.

Bibliography

[1] *Maude 2.0 Primer*. 51

[2] F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 1998. 99, 169

[3] J. Bang-Jensen and G. Gutin. *Digraphs*. Springer, 2000. 108

[4] T.M. Behrens, J. Dix, and K.V. Hindriks. Towards an environment interface standard for agent-oriented programming. Technical report, Clausthal University of Technology. 134

[5] M. Benerecetti, F. Giunchiglia, and L. Serafini. Model checking multiagent systems. Technical report, Instituto per la Ricerca Scientifica e Technologica. 33, 34, 35

[6] M. Benerecetti, F. Giunchiglia, and L. Serafini. Model checking multiagent systems. *Journal of Logic and Computation*. 33

[7] M. Benerecetti, F. Giunchiglia, and L. Serafini. Model checking multiagent systems. In *Intelligent Agents V*. 33

[8] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Tools and Algorithms for the Construction and Analysis of Systems*. 1999. 23

[9] R.H. Bordini, L.A. Dennis, B. Farwer, and M. Fisher. Automated verification of multi-agent programs. In *International Conference on Automated Software Engineering*, 2008. 33, 34, 35, 37, 38, 44

[10] R.H. Bordini, M. Fisher, C. Pardavila, W. Visser, and M. Wooldridge. Model checking multi-agent programs with casp. In *Computer Aided Verification*. 2003. 33

[11] R.H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking agentspeak. In *2nd International Joint Conference on Autonomous Agents and Multiagent Systems*, 2003. 3, 33

[12] R.H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Model checking rational agents. *Intelligent Systems*, 2004. 33

[13] R.H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. State-space reduction techniques in agent verification. In *3rd International Joint Conference on Autonomous Agents and Multiagent Systems*, 2004. 33, 34, 93, 94, 96

[14] R.H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifiable multi-agent programs. In *Programming Multi-Agent Systems*. 2004. 33, 107

[15] D. Bosnacki and S. Edelkamp. Model checking software: on some new waves and some evergreens. *International Journal on Software Tools for Technology Transfer*, 2010. 135

[16] M. Bramer. *Logic programming with Prolog*. Springer, 2005. 9

[17] J.R. Buchi. On a decision method in restricted second order arithmetic. In *International Congress on Logic, Methodology, and Philosophy of Science*, 1960. 144

[18] H. Kleine Buning and T. Lettmann. *Propositional logic: deduction and algorithms*. Cambridge University Press, 1999. 19, 141

[19] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. Technical report, Carnegie Mellon University, 1993. 41

[20] E.M. Clarke and E.A. Emerson. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 1982. 19

[21] E.M. Clarke, E.A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 2009. 20, 24

[22] E.M. Clarke, M. Fujita, S.P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. Technical report, Carenegie Mellon University, 1999. 94, 96, 107

[23] E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 1999. 110, 125, 129

[24] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. The MIT Press, 2000. 18, 19, 68, 69, 74, 84, 86, 87, 111, 112, 113, 114, 115, 116, 117, 119, 121, 122, 123, 139, 140

[25] E.M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Satabs: Sat-based predicate abstraction for ansi-c. *Tools and Algorithms for the Construction and Analysis of Systems*, 2005. 41

[26] M. Clavel, M. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J.F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 2002. 34

[27] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1992. 23, 38, 40, 51

[28] J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *FM'99 — Formal Methods*. 1999. 22

[29] M. Daniele, F. Giunchiglia, and M.Y. Vardi. Improved automata generation for linear temporal logic. 1999. 22, 140, 141, 142, 143

[30] F.S. de Boer, K.V. Hindriks, W. van der Hoek, and J.-J.Ch. Meyer. A verification framework for agent programming with declarative goals. *Journal of Applied Logic*, 2007. 7

[31] L. Dennis and M. Fisher. Programming verifiable heterogeneous agent systems. In *Programming Multi-Agent Systems*. 2009. 34, 36, 136

[32] L.A. Dennis. Agent infrastructure layer (ail): design and operational semantics. Technical report, University of Liverpool, 2007. 34

[33] L.A. Dennis, B. Farwer, R.H. Bordini, M. Fisher, and M. Wooldridge. A common semantic basis for bdi languages. In *Programming Multi-Agent Systems*. 2008. 34

[34] M. Dowson. The ariane 5 software failure. In *Software Engineering Notes*. 1997. 3

[35] M.B. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, Robby, and T. Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Tools and Algorithms for the Construction and Analysis of Systems*. 2006. 107

[36] C. Eisner and D. Peled. Comparing symbolic and explicit model checking of a software system. 2002. 23

[37] S. Eker, J. Meseguer, and A. Sridharanarayanan. The maude ltl model checker. *Electronic Notes in Theoretical Computer Science*, 2003. 34, 38, 41

[38] J. Esparza. Model checking using net unfoldings. In *TAPSOFT'93: Theory and Practice of Software Development*. 1993. 118

[39] K. Etessami and G.J. Holzmann. Optimizing buchi automata. In *CONCUR 2000 - Concurrency Theory*. 2000. 38, 40

[40] B. Farwer and L.A. Dennis. Translating into an intermediate agent layer: a prototype in maude. In *International Workshop on Concurrency, Specification and Programming*, 2007. 34

[41] A. Farzan and J. Mesequer. Partial order reduction for rewriting semantics of programming languages. *Electronic Notes in Theoretical Computer Science*, 2007. 36

[42] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *32nd Symposium on Principles of Programming Languages*. 118

[43] P. Gastin and D. Oddoux. Fast ltl to buchi automata translation. In *Computer Aided Verification*. 2001. 38

[44] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *15th International Symposium on Protocol Specification, Testing, and Verification*, 1995. 22

[45] D. Giannakopoulou and F. Lerda. From states to transitions: improving translation or ltl formulae to buchi automata. In *Formal Techniques for Networked and Distributed Systems*. 142

[46] P. Godefroid. *Partial-order methods for the verification of concurrent systems*. Springer, 1996. 31, 74, 86, 87, 111, 112, 117, 118

[47] P. Godefroid. Model checking for programming languages using verisoft. In *24th Annual Symposium on Principles of Programming Languages*, 1997. 23, 30

[48] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian partial-order reduction. In *Model Checking Software*. 2007. 118

[49] K. Havelund. Java pathfinder, a translater from java to promela. In *Theoretical and Practical Aspects of SPIN Model Checking*, 1999. 37

[50] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In *Automata, Languages and Programming*. 1980. 19

[51] K.V. Hindriks. Modules as policy-based intentions: modular agent programming in goal. In *Programming Multi-Agent Systems*. 2008. 7

[52] K.V. Hindriks. Programming rational agents in goal. In *Multi-Agent Programming*. 2009. 7, 19

[53] K.V. Hindriks, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. Agent programming with declarative goals. In *Intelligent Agents VII*. 2001. 3, 4, 7

[54] K.V. Hindriks and M.B. van Riemsdijk. Using temporal logic to integrate goals and qualitative preferences into agent programming. In *Declarative Agent Languages and Technologies VI*. 2009. 125, 169

[55] G.J. Holzmann. An improved protocol reachability analysis technique. *Software: Practice and Experience*, 1988. 31

[56] G.J. Holzmann. State compression in spin. In *3rd SPIN Workshop*, 1997. 31, 135

[57] G.J. Holzmann. Model checking software. In *Engineering Theories of Software Construction*. 2001. 98

[58] G.J. Holzmann. *The SPIN model checker*. Addison-Wesley, 2004. 18, 19, 20, 21, 23, 31, 100, 116, 119, 135, 145

[59] G.J. Holzmann and D. Peled. An improvement in formal verification. In *Formal Description Techniques*. 1994. 121

[60] G.J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *The SPIN Verification Systems*. 1996. 38, 40, 51, 112, 117, 123

[61] J.E. Hopcroft and R.E. Tarjan. Efficient algorithms for graph manipulation. Technical report, Stanford University, 1971. 121, 122

[62] D.A. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the IRE*, 1952. 135

[63] S.-S.T.Q. Jongmans, K.V. Hindriks, and M.B. van Riemsdijk. Model checking agent programs by using the program interpreter. In *Computational Logic in Multi-Agent Systems*. 2010. iii, 5, 33

[64] P. Kars. The application of promela and spin in the bos project. In *The SPIN Verification System*. 1997. 21

[65] O. Kupferman and M.Y. Vardi. Module checking. In *Computer Aided Verification*. 1996. 135

[66] O. Kupferman and M.Y. Vardi. Module checking revisited. In *Computer Aided Verification*. 1997. 135

[67] Dennis L.A, B. Farwer, R.H. Bordini, and M. Fisher. A flexible framework for verifying agent programs. In *7th International Joint Conference on Autonomous Agents and Multiagent Systems*, 2008. 34

[68] L. Lamport. What good is temporal logic. *Information Processing*, 1983. 113

[69] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *Model Checking Software*. 2001. 31, 37

[70] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *12th Annual Symposium on Principles of Programming Languages*, 1985. 20

[71] P. Linz. *An introduction to formal languages and automata*. Jones and Bartlett, 2006. 21, 42

[72] A. Lomuscio, W. Penczek, and H. Qu. Partial order reductions for model checking temporal epistemic logics over interleaved multi-agent systems. In *International Conference on Autonomous Agents and Multiagent Systems*. 118

[73] A. Lomuscio, W. Penczek, and H. Qu. Towards partial order reduction for model checking temporal epistemic logic. In *Model Checking and Artificial Intelligence*. 2009. 118

[74] K.L. McMillan. *Symbolic model checking*. PhD thesis, Carnegie Mellon University, 1992. 23, 118

[75] L.I. Millett and T. Teitelbaum. Issues in slicing promela and its applications to model checking, protocol understanding, and simulation. *International Journal on Software Tools for Technology Transfer*, 2000. 93, 94, 107

[76] M. Mukund. Linear-time temporal logic and buchi automata. In *Winter School on Logic and Computer Science*, 1996. 139, 144

[77] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. *Software Engineering Notes*, 1984. 94

[78] R. Pelanek. Fighting state space explosion: review and evaluation. In *Formal Methods for Industrial Critical Systems*. 2009. 130

[79] D. Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification*. 1993. 111

[80] D. Peled. Combining partial order reductions with on-the-fly model-cecking. *Formal Methods in System Design*, 1996. 111, 113

[81] D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 1997. 113

[82] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, 1977. 17, 20

[83] F. Raimondi. *Model checking multi-agent systems*. PhD thesis, University of London, 2006. 4, 33

[84] C. Robson. *Real world research*. Blackwell, 2002. 41, 42

[85] S. Russell and P. Norvig. *Artificial intellignece: a modern approach*. Prentice Hall, 2002. 106

[86] T. Schlipf, T. Buechner, R. Fritz, M. Helms, and J. Koehl. Formal verification made easy. *IBM Journal of Research and Development*, 1997. 20

[87] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 1993. 3, 7

[88] J. Slaney and S. Thiebaux. Blocks world revisited. *Artificial Intelligence*, 2001. 42

[89] F. Somenzi and R. Bloem. Efficient buchi automata from ltl formulae. *Computer Aided Verification*, 2000. 21, 22, 38, 40, 139

[90] H. Tauriainen. Nested emptiness search for generalized buchi automata. Technical report, Helskinki University of Technology, 2003. 23, 147

[91] H. Tauriainen. Nested emptiness search for generalized buchi automata. In *4th International Conference on Application of Concurrency to System Design*, 2004. 23, 25, 26, 147, 148

[92] H. Tauriainen. Nested emptiness search for generalized buchi automata. *Fundamenta Informaticae*, 2006. 21, 23, 40, 144, 145

[93] H. Tauriainen and K. Heljanko. Testing ltl formula translation into buchi automata. *International Journal on Software Tools for Technology Transfer*, 2002. 22, 139, 140

[94] W. Thomas. Automata on infinite objects. In *Handbook of theoretical computer science*. 1991. 20

[95] F. Tip. A survey of program slicing techniques. Technical report, Centrum voor Wiskunde en Informatica, 1994. 93, 94, 108, 109

[96] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1992. 74, 87, 115, 117, 123, 125, 129

[97] M.B. van Riemsdijk, L. Astefanoaei, and F.S. de Boer. Using the maude term rewriting language for agent development with formal foundations. In *To appear*. 2010. 34

[98] M.B. van Riemsdijk, F.S. de Boer, M.M. Dastani, and J.-J.Ch. Meyer. Prototyping 3apl in the maude term rewriting language. In *Computational Logic in Multi-Agent Systems*. 2007. 34

[99] M.Y. Vardi. Branching versus linear time: final showdown. In *Tools and Algorithms for the Construction and Analysis of Systems*. 2001. 19, 20

[100] M.Y. Vardi. Automata-theoretic model checking revisited. In *Verification, Model Checking, and Abstract Interpretation*. 2007. 18, 22, 23, 24, 38

[101] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *1st Symposium on Logic in Computer Science*, 1986. 20

[102] S. Vasudevan, E.A. Emerson, and J.A. Abraham. Efficient model checking of hardware using conditioned slicing. *Electronic Notes in Theoretical Computer Science*, 2005. 94, 96, 98, 107

[103] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 2003. 31, 33, 37, 93, 94

[104] M.D. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, 1979. 93

[105] M. Wooldridge, M.-P. Huget, M. Fisher, and S. Parsons. Model checking multi-agent systems with mable. In *1st International Joint Conference on Autonomous Agents and Multiagent Systems*, 2002. 33

[106] M. Wooldridge, M.-P. Huget, M. Fisher, and S. Parsons. Model checking for multiagent systems: the mable language and its applications. *International Journal on Artificial Intelligence Tools*, 2006. 33

[107] M. Wooldridge and N.R. Jennings. Pitfalls of agent-oriented development. In *2nd International Conference on Autonomous Agents*, 1998. 24

[108] J. Zhao, J. Cheng, and K. Ushjima. Slicing concurrent logic programs. In *2nd Fuji International Workshop on Functional and Logic Programming*, 1994. 93, 94