Modal Transition Systems as the Basis for Interface Theories and Product Lines

Nyman, Ulrik

Publication date:
2008

Document Version
Publisher's PDF, also known as Version of record

Modal Transition Systems as the Basis for Interface Theories and Product Lines

Ulrik Nyman

Aalborg University
Department of Computer Science

Modal Transition Systems as the Basis for Interface Theories and Product Lines

Modal Transition Systems as the Basis for Interface Theories and Product Lines

Ulrik Nyman

PhD Dissertation

October 3, 2008

Aalborg University
Department of Computer Science

Til Tina

Abstract

This thesis presents research taking its outset in component-based software development, interface theory and software product lines, as well as modeling formalisms for describing component based software systems and their interfaces.

The main part of the thesis consists of five papers. The first paper describes a framework for software product lines which can be instantiated for different design languages. Introducing the concept of *color-blindness* to describe the inability of an environment to distinguish the difference between several outputs, members of the software product line are automatically generated from a general model and the environment specifications. The next two papers each present an extension of Interface Automata. The first of these papers define Interface I/O Automata an interface theory for the I/O Automata community. The main novelty compared to previous work is an explicit separation of assumptions from guarantees and the presentation of a formally derived composition operator. The second of these papers present an interface theory combining Interface Automata and Modal Transition Systems into Modal I/O Automata. A formal correspondence between Interface Automata and a subset of modal transition systems is proved. The developed interface theory, which can describe liveness properties, is also applied as a behavioral variability theory for product line development. The two last papers of the thesis concern themselves with modal and mixed transition systems. The first of these present and discuss four different forms of consistency. Algorithms for synthesizing implementations from a given consistency relation is described for all four consistencies. The final paper proves PSPACE-hardness for *common implementation* and *thorough refinement* for mixed and modal transition systems. It also proves PSPACE-hardness for *consistency* of mixed specifications and establishes a number of reductions between the different decision problems.

Keywords: Modeling, Software Product Lines, Embedded Software, Modal Refinement, Labeled Transition Systems, Modal Transition Systems, Mixed Transition Systems, Modal Specifications, Mixed Specifications, Interfaces, Interface Theory, Interface Automata, I/O automata, Modal I/O Automata, Behavioral Inequalities, Consistency, Common Implementation, Thorough Refinement, Operational Characterization, Synthesizing Implementations, Relativized Simulation

ISSN nr: 1601-0590, Publication No. 45

Dansk sammenfatning

Denne afhandling præsenterer forskning der tager udgangspunkt i komponentbaseret software udvikling, grænseflade teori og software produkt familier, så vel som modellerings formalismer beregnet til at beskrive komponentbaserede software systemer og deres grænseflader.

Hoveddelen af denne afhandling består af fem artikler. Den første artikel beskriver en struktur for software produkt familier der kan instantieres for forskellige design sprog. Artiklen introducere konceptet *farve blindhed* (*color-blindness*) for at kunne beskrive en omgivelses manglende evne til at kunne skelne imellem forskellige output. Medlemmer af software produkt familien bliver automatisk genereret ud fra en general model og omgivelses beskrivelser. De næste to artikler præsenterer hver en udvidelse af Interface Automata (grænseflade automater). Den første af disse to artikler definerer Interface I/O Automata (grænseflade I/O automater) en grænseflade teori for I/O Automater. Den væsentligste nyskabelse i forhold til tidligere teorier er en eksplicit adskillelse af antagelser fra garantier og at den præsenterer en formelt udledt kompositions operator. Den anden af disse artikler præsenterer en grænseflade teori der kombinerer Interface Automata med Modale Transitions Systemer til Modal I/O Automata (modale I/O automater). Den indeholder også et bevis for en formal overensstemmelse mellem Interface Automata og en delmængde af modale transitions systemer. Denne grænseflade teori, der kan beskrive aktivitets (liveness) egeneskaber bliver også andvendt som en adfærdsmæssig variabilitets teori til software produkt familie udvikling. De sidste to artikler omhandler modale og blandede transitions systemer. Den første artikel præsenterer og diskuterer fire forskellige former for konsistens. Algoritmer der kan syntetisere implementationer ud fra en given konsistens relation bliver beskrevet for alle fire konsistenser. Den sidste artikel beviser PSPACE-hårdhed for *fælles implementation* (*common implementation*) og *grundig raffinering* (*thorough refinement*) for blandede og modale transitions systemer. Den viser også PSPACE-hårdhed for *konsistens* (*consistency*) for blandede specifikationer og fastslår en række reduktioner imellem de forskellige beslutnings problemer.

Nøgleord: Modellering, software produkt familier, indlejret software, modal raffinering, mærkede transitions systemer, modale transition systemer, blandede transition systemer, modale specifikationer, blandede specifikationer, grænseflader, grænseflade teori, grænseflade automater, I/O automater, modale I/O au-

tomater, opførselsmæssige uligheder, konsistens, fælles implementation, grundig raffinering, operationel karakteristik, syntesering af implementationer, relativiseret simulation

Acknowledgements

This thesis is a collection of five research papers that would not have been possible to write without the guidance and support from my supervisor Kim G. Larsen. Thank you for treating me like a colleague.

An equally big thank you to Andrzej Wąsowski, who in practice has functioned as a co-supervisor for me, for teaching me a lot through our close collaboration on all the papers in this thesis. And a thank you for keeping me motivated on reaching my deadline for handing in this thesis.

Also a thank you to Michael Huth and Adam Antonik for our new and fruitful collaboration, which I look forward to continuing.

Finally a very big thank you to Tina, my loving wife, for supporting me and truly believing in me. Thank you for bringing joy into my daily life. And a thank you to my son Mads for helping me take my mind off this thesis.

Contents

Introduction

1 Motivation

Software systems contain errors. Even though complex software systems have been developed for more that 30 years it is evident that systems being developed today still contain errors. For general purpose desktop software errors can cost money and time in the form of lost production time and man hours spent on software upgrades and failure recovery.

Why is it the case that software systems still contain errors despite the developments in software engineering practices that have taken place over the past 30 years? I will claim that a lot of the errors that exist in modern software systems are there for a very obvious reason: As a software system grows in size and complexity it becomes increasingly harder to avoid errors in the system. The software systems that are being constructed today are in general significantly more complex than those constructed 30 years ago.

One might argue that we simply have to learn to live with errors in software. One suggestion might be to work around the errors by providing a quick restart of the given system from a known correct initial state. For some types of systems this kind of approach is out of the question. This includes real-time systems that need to be continuously available and distributed networked applications where errors in the communication pattern could require a concurrent restart of software at many remote locations.

More and more software is being used in contexts where failure will have very serious consequences. The most serious consequence being the loss of human lives, be it from failing medical equipment or errors in air-traffic control systems.

What solutions could there be to avoid or reduce the number of errors in a given software system? I will in the following present two avenues which I think in combination can be a help in reducing the number of errors in large and complex software systems.

The first is *formal methods*: In order for methods to have a really profound impact on software production, and the number of errors in software systems, a given method has to lend itself to automation. Of course there are errors that exists due to misunderstandings during the specification stage. But even for such types of errors formal methods might have something to contribute as the use

of formal models during specification stage often will reveal inconsistencies and incompleteness of requirements.

The second part of the solution is to develop *component based* software systems: The idea is that, by dividing software systems into components each component will be smaller and less complex than the complete system. The challenge then lies in ensuring that the software components cooperate correctly. In order to truly reduce the complexity of the complete system, by splitting it into separate components, the interfaces between the components have to be clearly specified. If the interfaces are clearly and formally specified the correct operation of the complete systems can be achieved in two separate steps. Firstly it can be verified for each component that it behaves according to its interface specification. This can be done both at the design stages, through model checking, and later in the development process through testing. Model checking a detailed model of the component against its interface specification would ensure that the component satisfies the guarantees that the interface gives under the the requirements that the interface puts on its surroundings. Secondly one needs to establish, again through model checking, that the requirements and guarantees of the individual components together fulfill the requirements for the complete system.

Developing software systems based on components also opens the possibility of changing only some of the components in order to produce variants of the software systems. Such a collection of related products made from a common set of components is know as a Software Product Line. The combination of software product lines and formal methods is an emerging field of research. Given that software product lines are defined from a set of variant components the development method lends itself to formal methods. The different component variants can be combined in a myriad of ways and thus countless different configurations have to be tested if classic test based verification is used alone.

Formal methods have the possibility to allow for easier combination of already existing, and individually tested, software components by ensuring the compatibility of a given combination of components.

The research contained in this thesis is presented in the form of a collection of five articles. The articles are included in the chronological order in which they were written. The research grew out of work in software product lines. In this context we proposed the formalism of Color-blind transition systems, which can be used to express the dynamic inability of a given environment to distinguish between certain output. Descriptions of the environments as Color-blind transition systems could then be used to specialize and minimize a given software system to a specific context. From here the research progressed towards developing an interface theory based on I/O Automata. In the formal derivation of the composition operator for the interface theory, we used Modal Transition Systems. After this we extended Interface Automata with modalities into Modal I/O Automata. We also sketch a Software Product Line theory based on Modal I/O Automata. It later became clear that several interesting and quite fundamental

questions about Modal and Mixed Transition Systems remained open. The final two papers centers directly on questions regarding Mixed and Modal Transition Systems.

As written in the outline below the three fields of related work: Modal and Mixed Transition Systems, Interface Theories and Software Product Lines are presented in the reverse order in which I encountered them. It is presented in this way because it is beneficial to cover the most foundational work first and then build on top of this.

Outline

The rest of this introduction is organized in the following way.

The thesis first continues with a description of related work within three fields, namely: Modal and Mixed Transition Systems, Interface Theories and Software Product Lines. After this follows a summary of each of the five papers that make up the main part of the thesis. These summaries consists of a short abstract of the paper, its publication history and the contribution that it makes.

2 Modal and Mixed Transition Systems

This section contains an overview of related work within the field of Modal Transition Systems (MTS) and Mixed Transition Systems (MxTS). First we introduce the beginning of the field of research followed by sections on different topics.

Modal Transition Systems (MTSs) are extensions of Labelled Transition Systems (LTSs) with modalities on the transitions. MTSs have two transition relations describing respectively required and allowed behavior. We will later in this section define both Modal and Mixed transition systems. A formal definition of MTS and MxTS can also be found in Section 2 (p. 121) of Paper D as well as Section 3 (p. 145) of Paper E.

2.1 The Beginning

The field was started in 1988 with the paper *A Modal Process Logic* [LT88], which introduces Modal Transition Systems (MTS) as a way of specifying sets of implementations. The main argument used for introducing MTS is that Process Algebras will only allow for a certain degree of looseness when used as a specification language. This degree of looseness exactly corresponds to the observable behavior, such that all the different implementations are observationally equivalent. MTSs are introduced as a specification language which allows for looseness in the specifications. Figure 1 shows a simple example of a Modal Transition System.

Modal Transition System: For an action alphabet Σ, a *modal transition system* M is a triple $(S, R^{\Box}, R^{\diamond})$, where S is a set of states and $R^{\Box}, R^{\diamond} \subseteq S \times \Sigma \times S$ are must- and may- transition relations respectively. The transition relations of a *modal transition system* must satisfy the following condition $R^{\Box} \subseteq R^{\diamond}$, meaning that all its must-transitions are also may-transitions. A *pointed* modal transition system (M, s_0) is a modal transition system M with a designated initial state $s_0 \in S$.

In the original definition of modal transition systems from [LT88] the two transition relations were written as $\longrightarrow_{\Box}$ and $\longrightarrow_{\diamond}$. In the definition above and in Paper E we follow the original idea of using $\Box$ to represent necessary behavior and $\diamond$ to represent allowed behavior. In Paper D we have also adopted the convention of writing the *may* transitions using dashed lines ($\dashrightarrow$) and the *must* transitions using solid lines ($\rightarrow$). This convention is also used in the figures of Papers D and E. Using this notation instead of putting a symbol next to each transition allows for more compact and easily understandable figures.

The other central concept of the first paper [LT88] on MTSs is modal *Refinement*. Modal refinement captures what it means for one process to be more precise than another. One should note that modal refinement in most papers, including the original paper, is referred to simply as refinement.

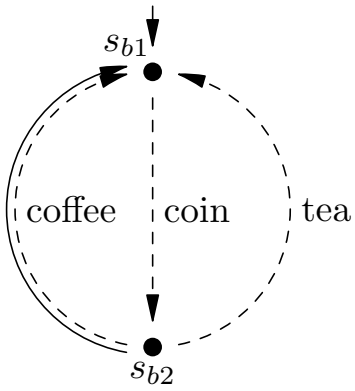


Figure 1: A simple Modal Transition System (MTS) with two states. The system describes a simple specification for a tea and coffee machine. In the initial state s_{b1} there is one outgoing may transition to state s_{b2}. This indicates that a given implementation can choose to implement this transition. From state s_{b2} there is an outgoing may transition labelled tea indicating that an implementation can choose to implement this transition. From s_{b2} there are also a may and must pair of transitions labelled coffee. This specifies that any implementation that first accepts a coin must be able to provide coffee.

The intuition behind modal refinement of MTSs is that one can refine one MTS M by changing some of the *may* transitions into *must* transitions and simply remove other *may* transitions and thus end up with a new system N which refines M.

Modal Refinement: A modal specification $(N, t_0) = ((S_N, R_N^{\Box}, R_N^{\Diamond}), t_0)$ *refines* another modal specification $(M, s_0) = ((S_M, R_M^{\Box}, R_M^{\Diamond}), s_0)$ over the same alphabet, written $(M, s_0) \prec (N, t_0)$, iff there is a relation $Q \subseteq S_M \times S_N$ containing (s_0, t_0) and whenever $(s, t) \in Q$ then

1° for all $(s, a, s') \in R_M^{\Box}$ there exists some $(t, a, t') \in R_N^{\Box}$ with $(s', t') \in Q$.

2° for all $(t, a, t') \in R_N^{\Diamond}$ there exists some $(s, a, s') \in R_M^{\Diamond}$ with $(s', t') \in Q$.

Figure 2 shows a set of MTSs and how they refine each other. A modal refinement relation Q witnessing that N refines M describes a form of recursive two way simulation where N must continually match the required behavior of M and M must continually match the allowed behavior of N. A modal refinement relation Q_1 witnessing that $s_{a1} \prec s_{b1}$ in Figure 2 would be $Q_1 = \{(s_{a1}, s_{b1}), (s_{a2}, s_{b2})\}$.

All systems obtained by the previous intuition, removing *may* transitions or changing them into *must* transitions, are legal by the formal definition. But the definition also allows for additional refinements, in that *must* transitions that have become unreachable due to the dropping of *may* transitions also can be

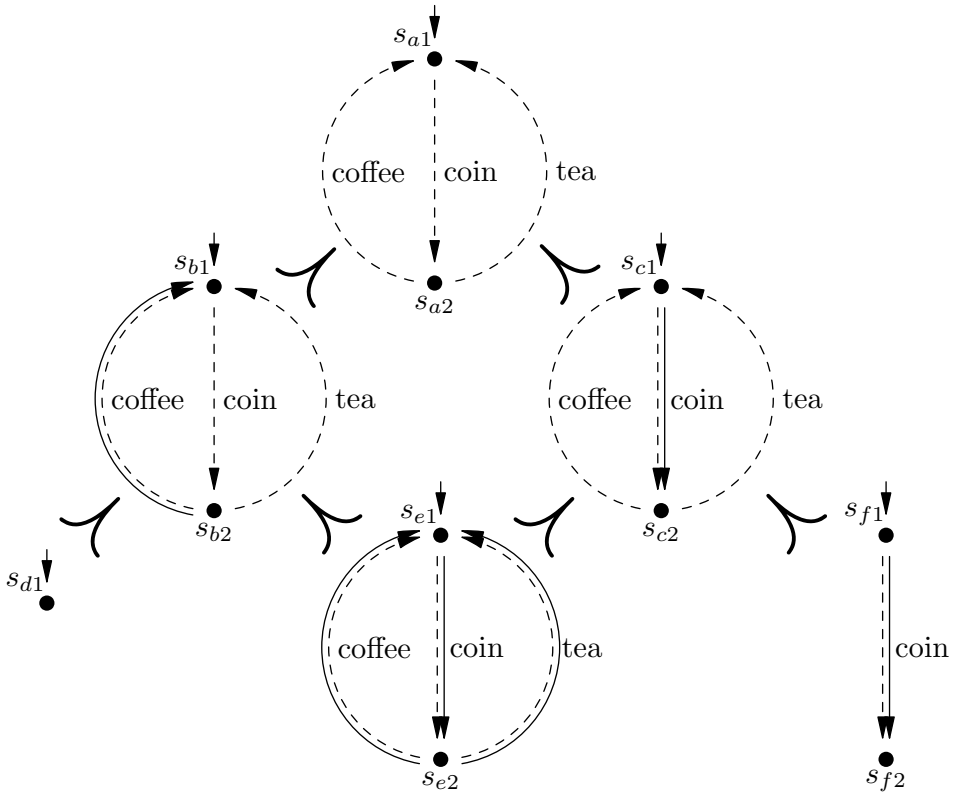


Figure 2: This figure explains modal refinement by example. The top most figure with the two states s_{a1} and s_{a2} is a very general specification of a coffee and tea machine. All the other MTSs are refinements of this specification. The refinement relations between the MTSs is given by the relations between the figures. The state s_{e1} is thus a refinement of both s_{b1} and s_{c1}. Notice that the single initial state s_{d1} refines s_{b1} even though the state s_{b2} has outgoing must transitions.

dropped. The relation between modal refinement and what it means for a single mixed transition system to be syntactically consistent is covered in Section 4 of Paper D.

Stepwise Refinement

Modal transition systems were originally conceived as being used in a stepwise refinement development process [Lar89], where systems are developed in a top down fashion. Starting with a specification that the systems should fulfill and slowly progressing to more and more detailed descriptions of the systems behavior, proving modal refinement for each step. Figure 2 illustrates how the top most general specification can be refined in different ways leading to more and more concrete specifications.

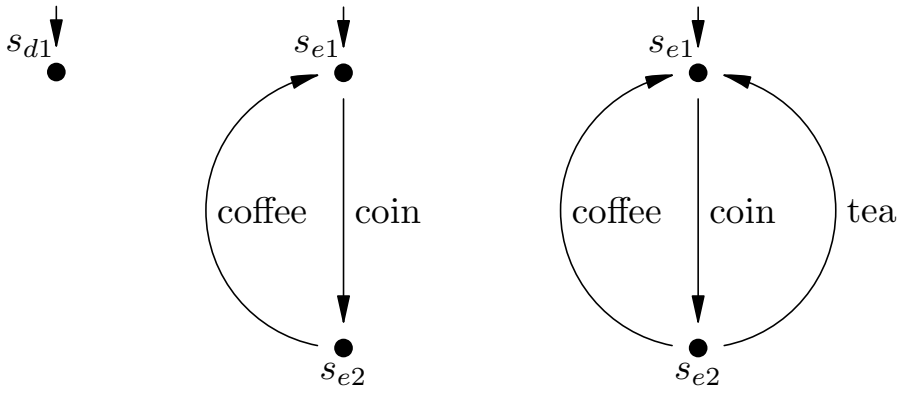


Figure 3: Three implementations, depicted as LTS, that each refine the MTS shown in Figure 1.

Implementations

When continuing to refine the specification a final implementation will eventually be reached. An implementation is characterized by having only one transition relation or equivalently two identical transition relations. Implementations are also known as processes. Implementations (processes) can thus be viewed as a special case of MTSs where the required and allowed behavior is identical, in effect making them identical to Labelled Transition Systems (LTSs). A process P is said to be an implementation of a specification S if it refines it. Figure 3 shows three simple, yet quite different, LTSs that refine the MTS in Figure 1.

2.2 Early Related Work

The 1989 paper *Modal Specifications* [Lar89] repeats the definitions of the original paper, presents a logical characterization of Modal Specifications and defines how to derive logical formulas characterizing a given Modal Specification. The paper also suggest ways of extending the expressive power of Modal Specifications with regards to liveness properties by making logical combinations of Modal Specifications.

In the 1990 and 1992 papers *Graphical versus Logical Specifications* Boudol and Larsen thoroughly treats the relationship between characteristic formulas in Hennessy-Milner Logic (HML) and Modal Transition Systems [BL92, BL90]. It is established that every MTSs can be characterized by a HML formula but only HML formulas that are consistent and prime, meaning that every disjunction implies one of the disjuncts, are graphically representable.

The 1989 paper *The Use of Static Constructs in a Modal Process Logic* [HL89] introduces the use of constructs from CCS [Mil89] such as parallel composition of components into Modal Process Logic (MPL). The paper also introduces unobservable transitions and an observational refinement relation. It investigates

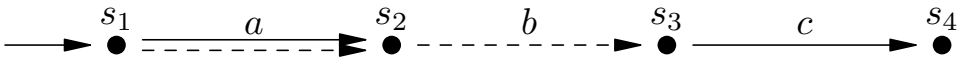


Figure 4: A simple Mixed Transition System (MxTS) where the state s_3 is inconsistent because it requires a c transition that is not allowed

under which constructs of MPL this observational refinement relations behaves as a congruence. The paper shows that MPL is, theoretically less expressive than Henessy-Milner Logic (HML) with recursion. But at the same time the paper demonstrates that, from a practical point of view MPL with static constructs suffices for modelling realistic systems. This is demonstrated through the modelling of a *data-link protocol*, a *restartable system*, and a simple *scheduler*. In all of these examples the looseness of the specifications, that MPL allows for, plays an important role. The paper also introduces, through the use of parallel composition, the possibility of unobservable τ actions and an observational refinement ordering as an extension of modal refinement.

2.3 Mixed Transition Systems

The original paper [LT88] only specifies modal transition systems which fulfill the stringent consistency condition that all *must* transitions should be matched directly by a *may* transition ($R^{\Box} \subseteq R^{\diamond}$). Systems fulfilling the condition will always have at least one implementation, having transition relations identical to the specifications *must* transition relation. Already in [Lar89] it was suggested to lift this consistency requirement in order to be able to model inconsistent systems. In [Lar89] and also in Paper D these systems are still referred to as modal transition systems. Such systems have later come to be known under the term Mixed Transition Systems (MxTS), as introduced by Dams [Dam96], and are also referred to as mixed specifications in Paper E. Mixed transition systems are more complex but are also much more interesting as a modelling formalism in that they allow for parallel composition of specifications that are mutually inconsistent. Figure 4 shows a simple Mixed Transition System in which the state s_3 is inconsistent. A MxTS in which the initial state is inconsistent will have no implementations, not even an empty system with only an initial state and no transitions.

Mixed Transition System: For an action alphabet Σ, a *mixed transition system* M is a triple $(S, R^{\Box}, R^{\diamond})$, where S is a set of states and $R^{\Box}, R^{\diamond} \subseteq S \times \Sigma \times S$ are must- and may- transition relations respectively. A *pointed* mixed transition system (M, s_0) is a mixed transition system M with a designated initial state $s_0 \in S$.

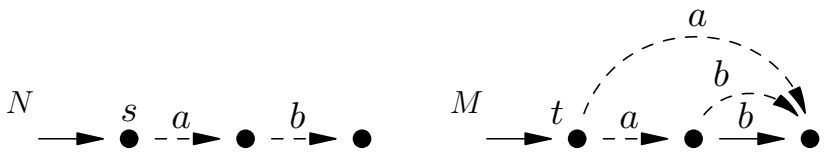


Figure 5: All implementations of N are also implementations of M but it is not the case that N is a modal refinement of M

2.4 Thorough Refinement

Thorough Refinement is a refinement relation based on inclusion of sets of implementations, such that one system refines another if all implementations of one also are implementations of the other. By refining a system, it is only possible to remove implementations and given two specifications N and M where the implementations of N are a subset of the implementations of M, it is the case that N thoroughly refines M. As shown in Paper E thorough refinement is PSPACE-hard to check, both for modal and mixed transition systems.

Modal refinement is sound but incomplete with regards to thorough refinement. Because it is sound we know that when a specification N refines a specification M then all implementations of N are implementations of M. By incompleteness we mean that there exists specifications N and M such that N does not refine M but it is still that case that all implementations of N are indeed implementations of M. The reason one might still want to use modal refinement instead of thorough refinement is that modal refinement is polynomial time decidable.

The fact that modal refinement is incomplete was first stated in the first piece of related work for the original paper, namely a master thesis *Operational and Denotational Properties of Modal Process Logic* [Hüt88]. An example demonstrating this, taken from [Hüt88], is given in Figure 5.

In the 2005 paper [Hut05b] it was stated, as the title says that *Refinement is complete for implementations*. This is unfortunately not the case as modal refinement is non-thorough. This was, as just stated, first reported in [Hüt88] and again in [Xin92], but was not reported in any conference or journal publication until [LNW07c] (Paper D) and is also to appear in the journal paper [SF07].

2.5 Modal Refinement Symbol

Several different symbols have been used for the concept of modal refinement. In order to avoid possible confusion we will in the following describe the different symbols and their opposing intuitions.

In the original paper [LT88] (and others) a $\lhd$ is used to indicate that N is a refinement of M by writing $N \lhd M$. The intuition here is that N is more specific

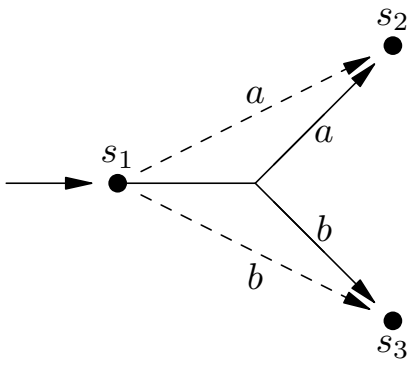


Figure 6: A simple Disjunctive Modal Transition System (DMTS) which requires that any given implementation will implement at least one of a and b. Both a and b are at the same time allowed.

than M, having less implementations and therefore the narrow end point towards N.

With the same intuition $N \leq_{\mathrm{m}} M$ is used in [LNW07a] (Paper C) and other papers to indicate that N has fewer implementations than M. This symbol also indicates that the two specifications might actually have exactly the same set of implementations.

In [AHL+08] (Paper E) the symbol $\prec$ is used with the opposite meaning that the narrow end points towards the side with more implementations. The intuition behind this is that the operator indicates which specification contains more information. The one that contains more information will be more restrictive towards allowing implementations and will possibly have fewer implementations.

Thus all the following statements have the same meaning:

Less implementations		More implementations
N	$\lhd$	M
N	$\leq_{\mathrm{m}}$	M
N	$\succ$	M

The $\leq_{\mathrm{m}}$ symbol is also used with different subscript than m for related forms of refinement. The $\prec$ operator also exists in a subscripted version $\prec_{th}$ signifying that the refinement is thorough.

2.6 Disjunctive Modal Transition Systems

In 1990 the concept of Disjunctive Modal Transition Systems (DMTS) was introduced in the paper *Equation Solving Using Modal Transition Systems* [LX90]. Disjunctive modal transition systems extend MTS by allowing a disjunction of

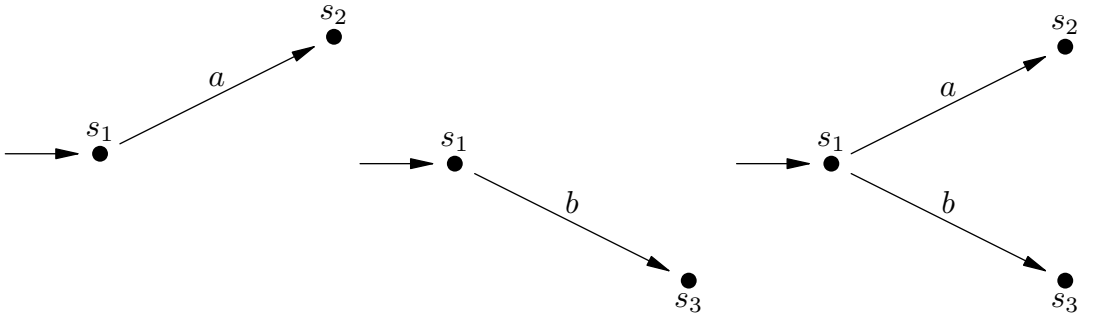


Figure 7: Implementations of the DMTS shown in Figure 6 depicted as LTSs. If the transition system in Figure 6 is interpreted as a 1MTS only the two left most transition systems are legal implementations.

target states and actions for each must transition. These transitions are also known as hypertransitions interpreted with an OR semantics. This signifies that a given implementation has the choice of implementing at least one of these branches. The modal refinement relation is extended accordingly.

Figure 6 shows a simple disjunctive modal transitions system. Three legal implementations of the DMTS in Figure 6 are show in Figure 7.

In [LX90] Larsen and Xinxin establish a method for solving process algebra equations involving contexts. A set of equations involving bisimulation equivalence $\sim$ [Par81, Mil83] and an unknown process X embedded in different contexts C_1 to C_n can be solved and the unknown process X can be expressed in the form of a DMTS.

This equation solving method is also the inspiration for the method for solving behavioral inequalities described in Section 5 of Paper B. DMTSs are also extensively covered in the PhD Thesis [Xin92].

In the 1991 paper *On the Complexity of Equation Solving in Process Algebra* [JL91] Jonsson and Larsen show PSPACE-hardness of solving process algebra equations involving contexts. The problems consists of deciding whether some process exists that, embedded in the given context, satisfies the given inequality or equality. The equations involve bisimulation, weak bisimulation and modal refinement. They also establish that given the context is deterministic, in the sense that one action from the same state cannot lead to different states, then some equations can be solved in polynomial time.

One Selecting Disjunctive Modal Transition Systems

One selecting Modal Transition Systems (1MTS) is a special version of DMTSs proposed by Schmidt and Fecher [Sch06b, SF07, Sch06a]. One selecting Modal Transition Systems interpret hypertransitions according to an XOR semantics such that precisely one of the target states can be implemented. In [SF07] Schmidt and Fecher show that DMTSs and 1MTSs can express the same sets

of implementations, but at the same time demonstrating that 1MTSs have a strictly more expressive refinement preorder compared to DMTSs. In the 2007 extended abstract *A behavioural model for product families* [FG07] Fantechi and Gnesi proposes a similar model which they call Extended Modal Labelled Transition Systems. The DMTS presented in Figure 6 can also be interpreted as a 1MTS, if this is the case only the two left most transition systems in Figure 7 are legal implementations.

2.7 Kripke Modal Transition Systems

In [HJS01] the extended modelling formalism Kripke Modal Transition Systems (KMTSs) is introduced. The paper first defines doubly labelled transition systems, which extend labelled transition systems and Kripke Structures by having labels on both transitions and states. KMTSs are then defined as an extension of both doubly labelled transition systems and MTS. In [Hut02] KMTSs are defined slightly different, as a composition of two doubly labelled transition systems, and termed as Modal Kripke Transition Systems. Here the two doubly labelled transition systems each represent both *must* labels and *must* transitions or *may* labels and *may* transitions respectively.

KMTSs generalises both Modal Transition Systems and Kripke structures. It has *may* and *must* transition relations as Modal Transition Systems and at the same time it has *may* and *must* labels on states. Huth et al. [HJS01] also extend the notion of refinement accordingly, putting requirements on the set of *must* and *may* labels of states that are related by refinement. In [HJS01] KMTSs are used as the basis for three valued program analysis and model checking of partial state spaces. The three values here being True (T), False (F) and Unknown ($\perp$).

In [Hut02] Huth proves that model checking of modal mu-calculus properties for KMTSs can be reduced to model checking of modal mu-calculus properties on regular Kripke Transition Systems. These results specialize to both CTL and CTL* model checking.

2.8 Timed Modal Specifications

In the 1993 paper *Timed Modal Specification - Theory and Tools* [ČGL93] the extended formalism of Timed Modal Specifications (TMSs) is presented along with the tool EPSILON for analyzing TMSs. TMSs is a generalization of real-time process calculi, over a dense time domain, with the aspect of allowing for loose specifications that Modal Transition Systems provide. The paper also defines timed refinement and several abstracting refinements. The abstracting refinements abstract away respectively internal computation, timing aspects and both in combination. These abstracting refinements are introduced to allow for more refinements by abstracting away concrete implementation details.

2.9 Applications

In the following we will present a number of areas and methods in which Modal and Mixed Transitions Systems have been used.

Automatic Abstraction and Model Checking

In the paper *A Constraint Oriented Proof Methodology Based on Modal Transition Systems* [LSW95] Larsen et al. introduce a framework for dividing the proof obligation for verifying a system through the use of abstraction, skolemization, separation of proof obligation and projective views. The method drastically reduces the complexity of relevant subproblems. In the 2001 paper *Abstraction-Based Model Checking Using Modal Transition Systems* [GHJ01] it is shown that one can check arbitrary formulas (existential as well as universal) using MTSs as abstractions at the same cost as checking universal properties using traditional conservative abstractions. The 2002 paper *Automatic Abstraction Using Generalized Model Checking* [GJ02] elaborates on how to utilize MTS for automatic abstraction in the framework of generalized model checking [BG00]. In the 2003 paper *On the Expressiveness of 3-Valued Models* [GJ03] it is shown that the complexity of generalized model checking does not change from one three valued formalisms to another.

Model Merging

The 2004 paper *Merging partial behavioural models* [UC04] and the 2006 paper *Properties of Behavioural Model Merging* [BCU06] proposes a framework for merging of behavioral models based on MTS. Model merging is based on observational refinement and merging two consistent models should result in their minimal common refinement. Model merging is intended for use in software engineering processes where scenario-based descriptions are elaborated iteratively. The utility of the method is demonstrated through a case study in [BCU06]. In [UBC07] MTSs are generated from a combination of Message Sequence Charts and safety properties expressed in Fluent Linear Temporal Logic (FLTL).

Product Lines

In the 2006 paper [FUB06] MTS are used as the foundation of a framework for Software Product Lines. This paper also presents a alternative branching semantics for Modal Transition Systems.

In [FG07] Fantechi and Gnesi propose a new extension of MTSs known as Extended Modal Labelled Transition Systems (EMLTS). This formalism is intended for definitions of Product Lines.

Industrial Application

In the 1997 paper *An industrial application of modal process logic* [Bru97] Bruns documents how Modal Process Logic was used in the development of a failure recovery protocol of an actual Air-traffic control system for use at Heathrow Airport. He also documents that CCS would have been inadequate for this application.

Trace Sets Semantics

In the 2001 paper *From Trace Sets to Modal-Transition Systems by Stepwise Abstract Interpretation* David Schmidt elaborates on how to transform execution trace sets, describing a systems semantics, into Modal and Mixed Transition Systems through stepwise abstract refinement. Universal and existential quantifiers can be added to CTL giving the logic mu-CTL* allowing for both trace and state checking of MxTS. He also proposes mu-XCTL* a variant of mu-CTL* specifically for expressing properties of Modal Transition Systems.

2.10 Tools

The 1992 and 1995 papers *Generality in design and compositional verification using TAV* [BLS93, BLS95] present the features of the TAV system. The TAV systems support for efficient decidability, stepwise refinement, and compositionality of MTSs is demonstrated through the analysis of a simple transmission protocol. TAV was later extended into the tool EPSILON for analyzing Timed Modal Specifications [ČGL93].

A plug-in for the Eclipse platform giving support for the construction, analysis and elaboration of MTS is presented in the 2007 paper [DFFU07].

3 Interface Theories

This section describes relevant related work within the field of interface theories.

The field of interface theories is rather new. It was started by the 2001 paper *Interface Automata* [AH01] by Alfaro and Henzinger and presented in an extended version in [AH04].

An interface theory is a behavioral type system for software components. Instead of only considering the static types of the components, an interface theory is concerned with the temporal aspects of the components, their behavior over time. In the original theory of *Interface Automata* the behavior of the individual components are described using an automata-based language. Using one automata per component the input assumptions and output guarantees of each component is described. The input assumptions describe the possible legal orders in which outside components can call methods on the component. The output guarantees describe the order in which the component may call other components. Since the two are described together in one automata, the output that a components produces is dependent on the input it has received. Similarly the input which it will allow is dependent on the output that it has itself produced.

They mention earlier work that considers automata based specifications of components [AG94, Lev95]. Their contribution is a both lightweight and formal approach to describing the behavioral interface of software components.

The main purpose of an interface theory is to check the compatibility of interfaces. In this way it is possible to infer whether the types of to components are compatible as is done with a traditional type system.

A novel aspect of Alfaro and Henzinger's composition operator is that it takes an *optimistic* approach. Instead of computing whether two given components will be compatible under any give circumstance, which would be the pessimistic approach, they compute under which circumstances the two components will be compatible. This means that two components are compatible if there exists some way to make them interact without violating each others assumptions. The result of computing a composition of two components will be a description of exactly how the two components can be used together.

They state that their formulation of component compatibility is related to [YS97] which considers synthesis of adaptors to bridge between incompatible interfaces. In this spirit they not only answer the question of whether there is some way of making the components interact but synthesize an interface representing all legal ways of using the new combined system.

Compatibility and Parallel Composition

An essential part of an interface theory is to be able to determine if components are compatible. As in many other theories Alfaro and Henzinger choose to define compatibility as a symmetric binary relation between two interfaces. Two

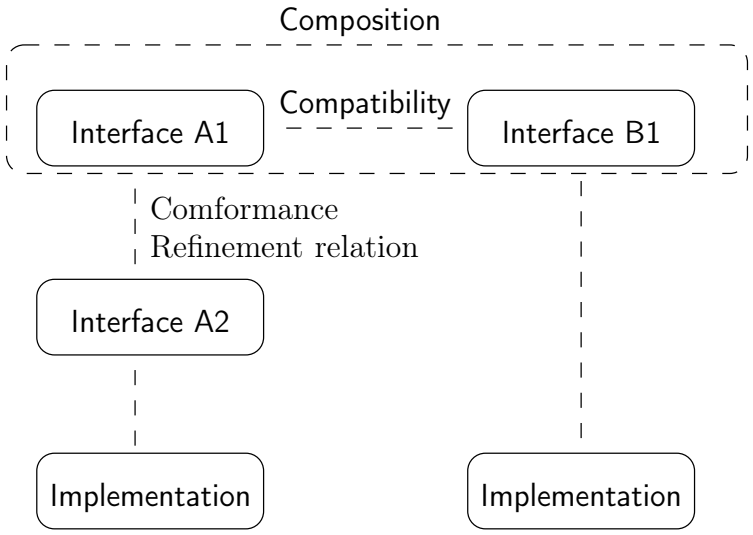


Figure 8: This figure gives an overview of the concepts: *Compatibility*, *composition*, *refinement* and *conformance*. *Compatibility* can be checked between interfaces, in this case Interface A1 and Interface B1. If the two interface are compatible their common interface can be computed, here represented by the dashed rectangle surrounding Interface A1 and Interface B1. Once compatibility has been established the *Independent Implementability* property ensures that Interface A1 can be refined into Interface A2 without considering Interface B1 at all. Each of the components can again be refined into implementations, conforming to the refinement relation.

interfaces are compatible if there is some way of using them together such that they do not violate each others assumptions. This can be said to be an optimistic approach or an existential view of compatibility in that the two interfaces need not be compatible in every setting, there just needs to be some setting in which they are compatible. Compatibility is the prerequisite for calculating the parallel composition of two interfaces. *Compatibility* is described using a binary symmetric relation between two interfaces, such that we write $F \sim H$ if F and H are compatible.

Once it has been established that two interfaces are compatible the *parallel composition* of the two can be calculated. The parallel composition, written $F \parallel H$ is essentially computed by constructing the product and pruning error states and internally controllable transitions that can reach error states. In this way the parallel composition describes the interface by which one can use the combined components and be sure that they cannot reach an error state. Figure 11 illustrates both compatibility and parallel composition.

Incremental Design and Independent Implementability

Henzinger and Alfaro state two requirements [AH04] for an interface theory, namely *Incremental Design* and *Independent Implementability*. *Incremental Design* is related to the concept *Stepwise Refinement* as presented in Section 2. The general idea is that the design process can start with a general description of each component and slowly refine the individual descriptions allowing for fewer possible implementations. During the incremental design of a system some components can be left out entirely. The other components will be seen as compatible if there just *exists* some component that can fill the empty place.

Incremental Design can be stated formally as a property. This property describes that if a closed system consisting of four components that are compatible in one setup, then the system should be composable in any order and the components should still be compatible.

If two components are compatible then the parallel composition $F||G$ of F and G is defined and represents the new common interface of the two components. Given this *Incremental Design* can be defined as follows:

Incremental Design [AH04]:
For all Interfaces F, G, H and I if
$F \sim G$ and $H \sim I$ and $F \parallel G \sim H \parallel I$
then
$F \sim H$ and $G \sim I$ and $F \parallel H \sim G \parallel I$

The other requirement for an interface theory is that of *Independent Implementability* which first appeared in the literature in [SDH00] where it appears in the context of formal models of Bus protocols specified as monitors. In [SDH00] independent implementability is only defined by a textual description and an example of what it is not. The idea is that requirements which should be checked on the components cannot have statements like $a = b$ where a and b are outputs of two components. If such a requirement is not fulfilled then one cannot blame one of the components, plus it is impossible for one component to be implemented without having complete information about the specific implementation of the other component. Thus such a requirement clearly violates independent implementability. As in the previous section on Modal Transition Systems, we define a binary refinement relation $\succeq$, in this case to describe that one interface refines another. This relation is defined as a contravariant simulation, know as *alternating simulation* [AH01], such that if we have $F \succeq F'$ (F is refined by F') then F' is able to provide more services than F but it must be consistent with F on the shared services. This can also be expressed slightly simplified, ignoring hidden transitions, as: An interface automaton F' refines an interface automaton F if each input transition of F can be simulated by F', and each output transition of F' can be simulated by F [AH04].

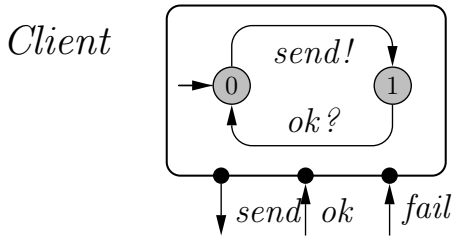


Figure 9: A simple *Client* component specified as an Interface Automata. This example is taken from [AH04]. Transitions labelled with ! represents output and transitions labelled with ? represent input. The transitions connected to the box around the component represent the static signature of the component. The fact that the component does not have any input transitions labelled by *fail* means that even thought *fail* is part of the static signature of the component it is never able to receive a fail message.

Figure 10: The two components in this figure, *Client* on the left and *FailingLink* on the right are incompatible, meaning that there is no way in which they can cooperate. They are not compatible because *FailingLink* sends a *fail* message in its initial state which *Client* is unable to receive.

Using these definitions *Independent Implementability* can be defined as follows.

Independent Implementability [AH04]:
For all interfaces F, F', G and G' if
$F \sim G$ and $F \succeq F'$ and $G \succeq G'$
then
$F' \sim G'$ and $F \parallel G \succeq F' \parallel G'$

Note that in contrast to the refinement relations presented in the previous section, the refined interface is allowed to add new services, but at the same time the set of possible implementations is decreased because now the specification of the newly added services have been fixed. This is in contrast to modal refinement where something would have to explicitly be modelled as allowed in the more general specification before the refined specification can specify how these services should be implemented.

Figure 9 presents a very simple *Client* component taken from [AH04]. If two components are compatible then it is possible to compute their composition,

which Alfaro and Henzinger write $F \parallel G$. Compatibility is determined by computing the product automaton $F \otimes G$ and checking if it contains error states that are reachable by internally controllable actions. If this is the case the two components are incompatible. Figure 10 shows two Interface Automata that are completely incompatible. If error states exists that are only reachable by externally controllable actions, then these are pruned from the product automaton in order to end up with the composite interface $F \parallel G$. In this way Interface Automata achieve that restrictions of one component can be propagated to the common interface when two components are composed. Figure 11 shows an example, taken from [AH04], where the dynamic ability of one component to receive two messages of the same type in a row will carry over to the combined interface.

Independent implementability gives rise to a form of subtype theorem, in that an interface automaton F always can be replaced by a more refined interface automaton F', such that $F \succeq F'$ provided that F and F' are connected to the environment by the same inputs. The side condition is there to ensure that F' cannot create new incompatibilities, by offering new functionality that the environment would use.

Earlier Work in Behavioral Type Theory

In a separate tradition in the field of Type Theory people have also tried to tackle the aspects of behavior, both of components and individual data elements, such as objects.

In the 1994 paper *A Behavioral Notion of Subtyping* [LW94] Liskov and Wing starts out by asking: What is subtyping?

They answer this question themselves by defining a subtype requirement: Let $\varphi(x)$ be a property provable about objects x of type T. Then $\varphi(y)$ should be true for objects y of type S where S is a subtype of T [LW94]. They only consider safety properties, *something bad will never happen* and not liveness properties *something good will eventually happen*. They consider two types of safety properties: *Invariants* and *history properties*. They mention other types of safety properties which they do not consider, e.g: the existence of an object in a state, the number of objects in a state, or the relationship between objects in a state. Invariants are defined as predicates on individual states, that must hold for all states. History properties are defined as predicates over pairs of states.

Liskov and Wing introduce the concept of substitutability, described in their abstract by the following:

> The use of hierarchy is an important component of object-oriented design. Hierarchy allows the use of type families, in which higher level supertypes capture the behavior that all of their subtypes have in common. For this methodology to be effective, it is necessary to have a clear understanding of how subtypes and supertypes are

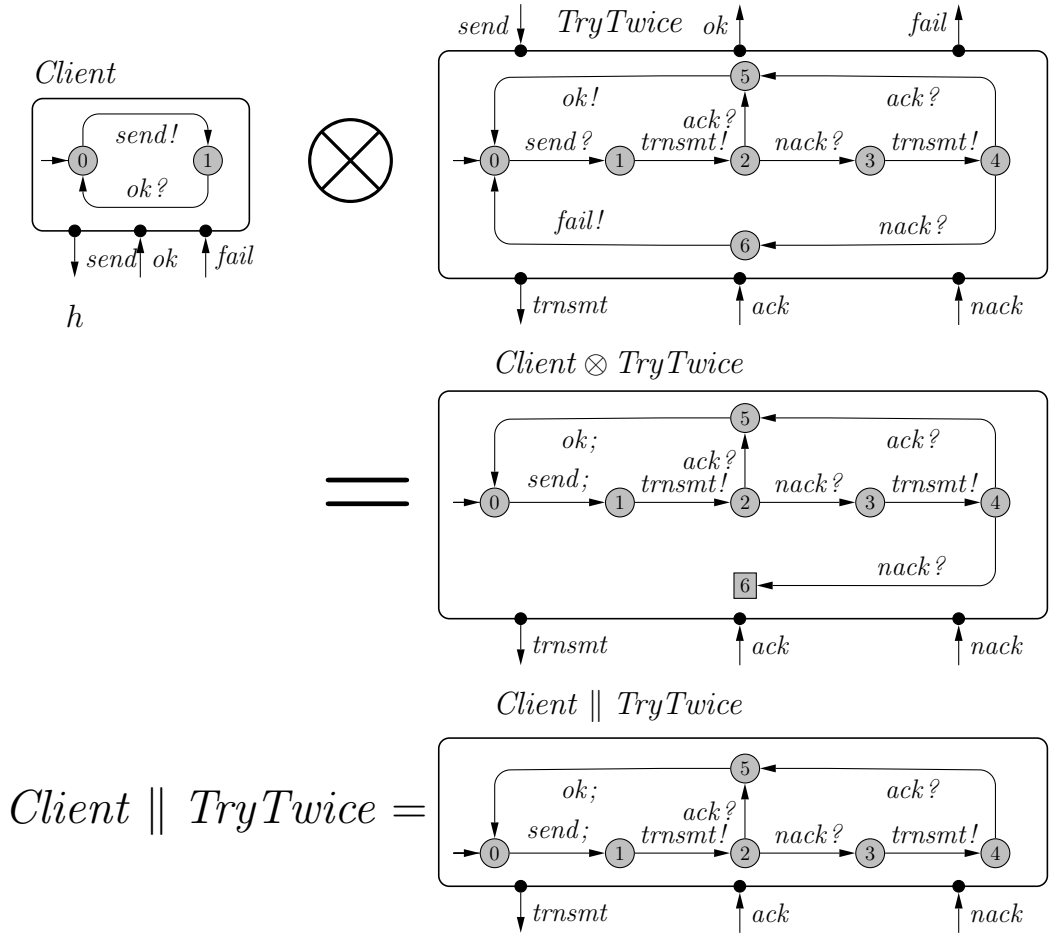


Figure 11: This figure demonstrates the concepts: *Composition* and *propagation of restrictions*. The two Interface Automata *Client* and *TryTwice* are combined into the product automaton *Client* $\otimes$ *TryTwice*. The bottom shows the composite interface *Client* $\|$ *TryTwice* that is computed by removing transitions such that no error states are reachable by internally controllable transitions. The square state in *Client* $\otimes$ *TryTwice* represents an error state, because *TryTwice* wants to output a *fail* message that *Client* cannot accept. By removing an externally controllable transition on the path to this state, the requirement that the *Client* cannot receive a *fail* message is propagated such that *Client* $\|$ *TryTwice* cannot receive a *nack* message twice in a row from its environment.

> related. This paper takes the position that the relationship should ensure that any property proved about supertype objects also holds for its subtype objects. [LW94]

The notion of substitutability is also known as the *Liskov Substitution Principle* within the field of type theory.

They propose two forms of subtyping. For both forms *invariants* are handled explicitly by stating an invariant property as part of the type definition. All

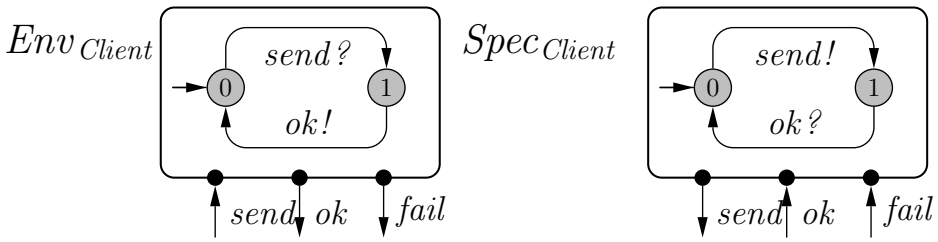


Figure 12: A simple *Client* component specified as an Interface I/O Automata, where the component is specified by the two parts Env_{Client} and $Spec_{Client}$. The explicit split of assumptions and guarantees allows for easy recombination of assumptions and guarantees from different settings.

subtypes must then fulfill this invariant. The two different notions of subtyping differ in how they handle *history properties*. The first type deals with history properties directly by requiring that a **constraint** is specified that states which types of history properties must be preserved by all subtypes. The second type of subtyping deals with *history properties* indirectly by requiring that for each new method added to a subtype its functionality is explained in the form of functions that already exist on the super type. This ensures that the new method does not introduce any genuinely new functionality.

Another more recent development is the Java Modelling Language (JML) [JP01, LBR06] which is a behavioral interface specification language for Java with invariants and pre- and post-conditions based on ideas from Eiffel [Mey92] and Larch specifications [GHW85]. Larch is a two-tiered modelling language where one level, known as Larch Shared Language, is programming language independent, and the other, known as interface languages, is specific to the given programming language that is being modelled [Win87]. Among many other approaches to behavioral specifications Leavens have adapted Larch specifications specifically to C++ Modules in [Lea96].

There is an apparent lack of references to work from the other tradition between the two directions of research, and a closer study of the relationship between Interface Theories and Behavioral Type Theory is called for.

Interface Input/Output Automata

In Paper B we present an interface theory fashioned for the I/O Automata community. The contributions of this paper is that it provides an interface theory for the I/O Automata community. The construction of an interface theory in an input enabled setting is achieved through an explicit separation of assumptions and guarantees. Figure 12 shows the simple *Client* component modeled as an Interface I/O Automata.

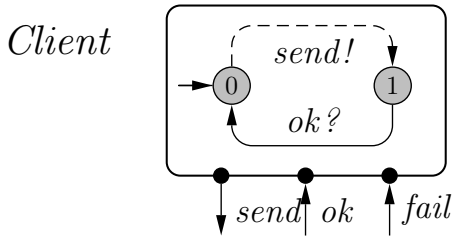


Figure 13: A simple *Client* component specified as an Modal I/O Automata. In contrast to the two other Client components shown in Figure 9 and Figure 12 this component will not allow trivial implementations that just do nothing.

Modal I/O Automata

In Paper C we show that interface automata corresponds to a subset of modal transition systems, and that on this subset the two notions of alternating simulation and modal refinement coincide. We define modal I/O automata as an extension of interface automata with modalities. The novel aspect of the interface theory that we build from modal I/O automata is that it can express liveness properties, such that trivial implementations that exhibit no behavior at all can be disallowed. The problem of trivial implementations exists for interface theories build on simulation preorders. Figure 13 shows the simple *Client* component modeled as a Modal I/O Automata.

In the following we will present several directions of research, each extending the interfaces with some new form of information.

Other Interface Theories

Alfaro et al. have in *Sociable Interfaces* [dAdSF$^+$05] extended interface automata with a more rich synchronisation scheme allowing for one-to-one, many-to-one, one-to-many and many-to-many communication as well as communication over shared variables.

Within the field of telecommunication Carrez et al. in the 2003 paper *Behavioural Contracts for a Sound Composition of Components* [CFN03] present an interface theory for the telecommunications industry. They are also inspired by Alfaro and Henzinger's interface automata and Larsen and Thomsen's modal transition systems. They place themselves in the tradition of process algebras. They have a slightly different interpretation of the *must* and *may* modalities, in that they talk of messages. In their framework a receiving port can, by having an input must action, *require* that the other component sends a message of a given type. They also, like our work in Paper B and C, are able to specify liveness properties.

In the 1997 paper *Specification matching of software components* [ZW97] Zaremski and Wing tackle four different concepts related to *component matching*. They describe component matching as the process of determining if two software

components are related. They see the four different but related issues: Retrieval, reuse, substitution and subtype. They define the four general questions relating to these concepts as:

Retrieval: How can I retrieve a component from a software library based on its semantics, rather than on its syntactic structure?

Reuse: How might I adapt a component from a software library to fit the needs of a given subsystem?

Substitution: When can I replace one software component with another without affecting the observable behavior of the entire system?

Subtype: When is an object of one type a subtype of another?

The two last questions are directly related to the concepts considered part of an interface theory by Alfaro and Henzinger. The other two concepts on the other hand are envisioned in a context where a developer already has access to a large library of components. Given that the developer has access to the source code of these components it would be extremely relevant to find components that are a near match for the desired functionality, which can be modified to fit the specific purpose. The descriptions of the components are given in Larch/ML, the version of Larch specifically targeted at the programming language ML.

In the 2006 paper *Component substitutability via equivalencies of Component-interaction automata* [ČVZ06] Cerna et al. formally characterize preconditions that lead to reconfiguration correctness of systems specified using Component-interaction automata [BvVZ05]. Reconfiguration correctness is defined as proper substitution and safe independent implementation. They characterize three relations which can be used to compare the behavior of two components with regards to reconfiguration correctness. Namely; the equivalence relation, the specification-implementation relation and the substitutability relation.

In the 2002 paper *Behavior protocols for software components* [PV02] Plasil and Visnovsky present a framework for adding behavior protocols, specified in the form of regular expressions to the SOFA architecture modelling language.

In the 2005 paper *Causality Interfaces and Compositional Causality Analysis* [LZZ05] Lee et al. create a framework for analysing causality between components seen as actors. This is similar in spirit to the work of Alfaro and Henzinger. They define an interface theory which represents the causality between actors and can compute the composed causality interface when composing components. They apply their framework to two actor-oriented models, namely synchronous languages and discrete event models.

In the 2003 paper *Resource Interfaces* [CdAHS03] paper Alfaro and Henzinger together with the coauthors Chakrabarti and Stoelinga extend their own framework to cover consumption of continuous resources, exemplified by the power consumption of components. Resource interfaces are modelled as games having quantitative objectives. The framework can help answer questions like; can

two components achieve their objectives without ever consuming more than the available peak power level.

In the 2002 paper *Conformance Checking for Models of Asynchronous Message Passing Software* [RR02] Rajamani and Rehof study the concept *stuck-freeness*, ensuring that programs do not get stuck trying to send or receive a message in vein.

Timed interfaces

In the 2002 paper *Timed Interfaces* [AHS02] the foundations of an automata based timed interface theory is crafted. This paper does not provide a complete timed interface theory in that it does not prove any independent implementability property.

Unlike most of the other research on interface theories, which is based on automata the following four papers on timed interface theories all base themselves on stateless assume/guarantee reasoning. The fact that the first paper on Timed Interfaces did not prove independent implementability and all the subsequent research is based on stateless formalisms, one might conjecture that it might be impossible to achieve independent implementability with the combination of automata and real-time.

In the 2005 paper *Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling* [WT05] Wandeler et al. introduce Real-time Interfaces that connects the principles of Real-time calculus with component based design.

In the 2006 paper *Interface-Based Design of Real-Time Systems with Hierarchical Scheduling*, [WT06] this approach is extended to real-time systems with a hierarchical mix of dynamic and static scheduling.

In the 2006 paper *An Interface Algebra for Real-Time Components* [HM06] Henzinger and Matic present an interface theory for real-time systems. They prove both incremental design and independent implementability.

In the 2006 paper *Real-time interfaces for composing real-time systems* [TWS06] Thiele et al. aim a unifying different approaches to real-time interfaces. They prove refinement and independent implementability and state the requirements that exist to a given framework for these properties to be fulfilled. They also introduce the novel notion of adaptive interfaces, which support features like modelling of end-to-end delays, buffer spaces and energy consumption. The applicability of the presented framework is demonstrated by applying it to a number of real-time analysis models.

Other Applications

In the 2005 paper *An Interface Theory Based Approach to Verification of Web Services* [CWD+06] Chen et al. present a framework for verification of web ser-

vices in which different properties can be verified at different levels of abstraction. The web services are described as Web Service Interfaces [BCH05]

In the 2006 paper *An algebraic theory for behavioral modeling and protocol synthesis in system design* [TG06] an interface framework for the multiple-clocked synchrony of the chip design world is defined.

Tools

The theory of interface automata is implemented in the tool CHIC which seems to no longer be actively developed. It has been replaced by the tool TICC [AdAdS+06] which implements compatibility checking and composition of interfaces from the more expressive interface theory of Sociable Interfaces presented in [dAdSF+05].

This concludes my chosen coverage of related literature within the field of behavioral interface theory. I have tried to include everything that has been published which is directly inspired by Interface Automata [AH01]. The coverage of literature related to behavioral type systems is much more sparse, mainly due to the fact that this filed of research seems to be both bigger and more diverse. A closer study of the relation between the two fields could be relevant future work.

4 Software Product Lines

The term Software Product Line (SPL) is used to describe a set of related software products with many similarities but varying functionality. SPLs are also know as software product families and in this context the individual software products are referred to as family members. Such families of software products arise in many different contexts. Two main examples are Enterprise Resource Planning (ERP) Systems and embedded software for families of related hardware devices, such as mobile phones. The process of managing the development of SPLs is generally referred to as software product line engineering.

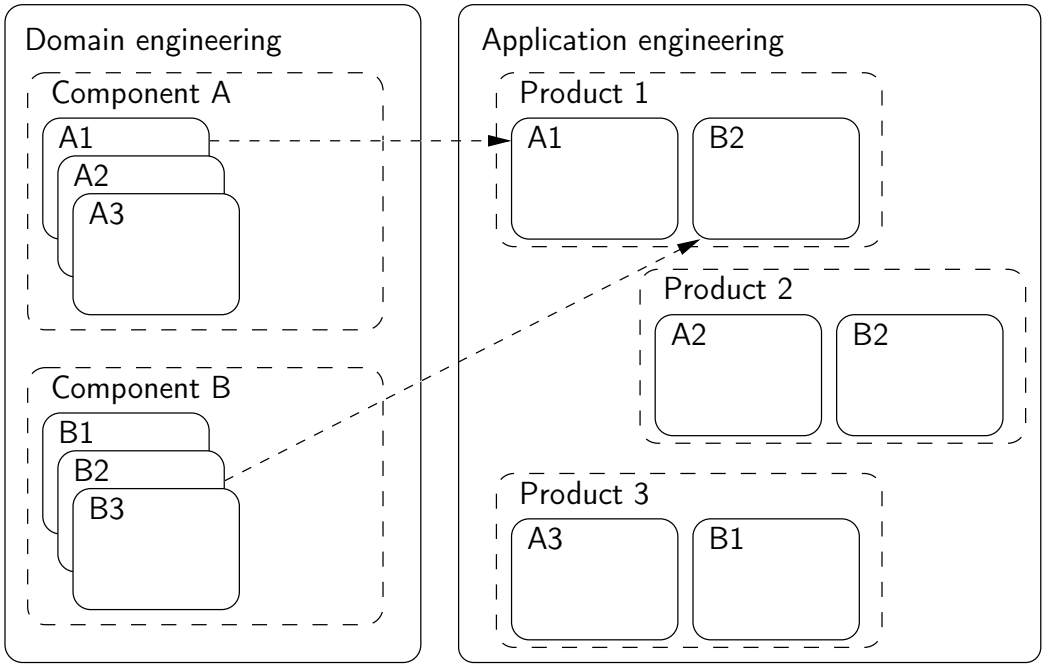


Figure 14: Figure giving an overview of a SPL setup. On the left: *Domain engineering* where general configurable components (assets) describing the functionality in the domain are created. On the right: *Application engineering* where the components (assets) are used to create specific products.

The software product line engineering process is generally divided into two separate processes: Domain engineering and application engineering. Software product line engineering is based on heavy reuse of the same components in different software products within the same line of products. It sets itself apart from normal software reuse in the way the reusable components are developed. They are not created in an ad hoc style where they are created specifically for one product first and then later adapted for general reuse. Through the process of *domain engineering* they are created with the original intent of being gen-

eral components that can be reused in several software products. These general components are also known as assets. In this process the intent is to capture all common functionality and group it together as components that can be used as building blocks. The process of creating an individual product of the SPL is known as *application engineering*.

The field of research concerning SPLs is very diverse in that it stretches all the way from software engineering topics such as requirements engineering [BHLP06, KE03], concrete industrial case studies [JB02, Bos99] over new development paradigms [BFK+99] to formal methods. A general overview of *Software Product Line Engineering* can be found in [PBvdL05]. In this section we will only cover the combination of SPLs with formal methods and in particular focus on behavioral models.

Feature Models

A feature model is a formalism for describing variability and commonality in a software product family. Many different feature models have been proposed in the literature [KCH+90, CHE05, Bat05]. A very simple feature model is presented in Figure 15. A feature model describes both what features remain fixed for every member of the product family and which features are variation points.

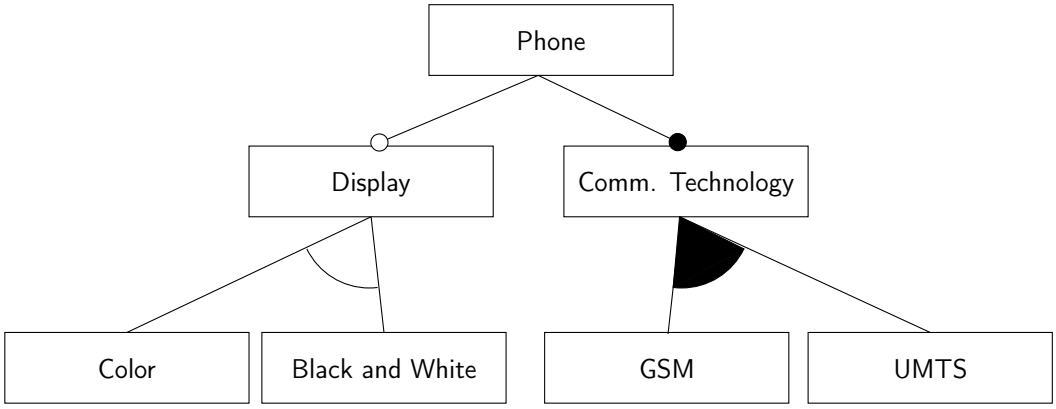


Figure 15: Figure showing a very basic feature model for a phone product line. The model describes that it is optional for the phone to have a display but required to have some form of communication technology. It also shows that in this particular feature model it is only possible to have exactly one type of display out of two choices. On the other hand it is allowed to have one or two forms of communication technologies.

In the 2004 paper [JB04] Jaring and Bosch present a formalization of variability dependencies in SPLs. Through a case study in designing a program monitor

and exception handler they show that the robustness of a SPL architecture is related to the type of variability.

Feature models can be translated into propositional formulas. In the 2007 paper [CW07] Czarnecki and Wąsowski describe a method for doing a form of reverse engineering in extracting feature models from propositional formulas.

4.1 Behavioral Models

The following section describes our research and related work within the field of behavioral models for software product line development. The behavioral models differentiate themselves from general feature models, in that they describe not only the structure of which components can be combined, but also the behavior of the individual components.

Color-blindness

In Paper A we present an SPL framework build around formal models and model transformations. The formal models that we use are State/Event Systems a variant of statecharts [Har87]. The research is a behavioral extension of the static concepts presented in [Wąs04]. The general structure of the framework is shown in Figure 16. In our framework the domain engineering process consists of creating one model that incorporates all the functionality that any product in the product line is going to have. The individual product is then constructed by first specifying an environment description for the given product. Secondly the general model is then transformed according to the environment. Thirdly the actual source code for the product is auto generated from the transformed model. Automated code generation from the similar formalism is covered in [Wąs03, Wąs05]. One approach to SPL engineering is to combine it with automated code generation from formal modals of the software that should be created.

Because all functionality is organized in one model the approach might, in its current form, be less suited towards larger product families. The general idea on the other hand is not restricted in this way. The general model might equally well be a collection of components that can be combined. The novel aspect of the theory is that the environments describe the inability of the environment to distinguish various output. This inability is dynamic in such a way that one might specify that the environment cannot distinguish two different lamps flashing, but only in the case where button B is pressed and not when button A is pressed. Paper A gives an example of a product line of alarm clocks in which, as one of the examples, different levels of lighting cannot be distinguished. This environment model represent the fact that the developer intend to only use one type of light bulb in this cheaper model. By specifying this inability, the model can be specialized and very likely become smaller. Thus resulting in a smaller code size for the given product.

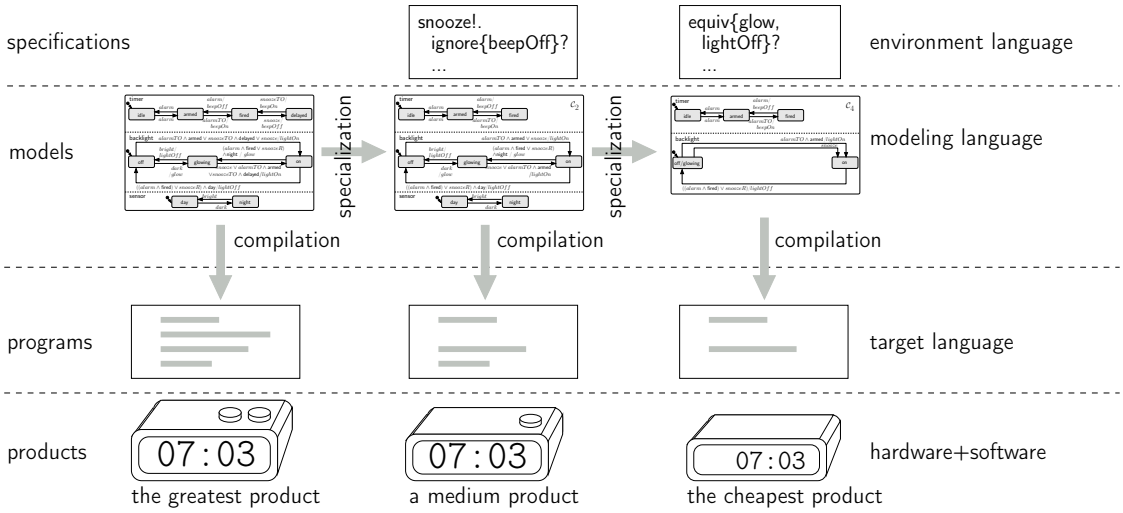


Figure 16: Overview of a product line of alarm clocks. The top left model is the general State/Event System model specified by the developer. The developer has also specified the usage descriptions in the form of environments, above the two rightmost columns. This figure is the same as Figure A.1.

Modal I/O Automata

In Paper C we present a behavioral variability theory for product line development. This work is closely linked to the interface theory also presented in Paper C. The components are again specified as Modal I/O Automata. In this case the Modal I/O Automata are not only descriptions of the interface of the components that may later be implemented. They are now descriptions of already existing configurable components. Each model describes a family of component variants. For the interface theory the type of question that was answered was of the type: What is the composite interface of two components such that all possible implementations of the two interfaces will always function together? In the product line theory based on Modal I/O Automata we ask the question: Is there at least a pair of implementations of the two components such that these two implementations will function correctly together? The outcome of this question is an interface describing all products that can be created from the components at hand. Given an interface that specifies the desired component it is now possible to check refinement between this interface and the model of all possible products. In this way one can determine if the desired product can be built by configuring already existing assets.

Extended Modal Transition Systems

In the 2007 extended abstract *A behavioural model for product families* [FG07] Fantechi and Gnesi proposes the new model called Extended Modal Labelled Transition Systems (EMLTSs). They transform Product Line Use Cases (PLUCs) into EMLTSs. The internal consistency of a family of products defined as an EMLTS can be verified as well as the family membership of a specific product, specified as a LTS, can be checked via a refinement relation. The complexity of checking this refinement relation is not discussed. Citing [LNW07a] they state that their novel aspect in comparison to our work is the ability to specify that at least k of n possible transitions should be chosen at a particular point. A sub case of this is the common scenario where exactly one of a set of components should be chosen.

In the 2006 paper *A foundation for behavioural conformance in software product line architectures* [FUB06] Fischbein, Uchitel and Braberman define a novel conformance relation to check whether a LTS is an implementation of a given modal transition system. The conformance relation, which is called *Branching Implementation* is established via a fix-point algorithm. The conformance relation does not apply directly to checking whether one interface is a refinement of another, for this purpose they suggest a version of thorough refinement where one interface refines another if all implementations of one are also implementations of the other. They hint that this relation might be hard to establish algorithmically and based on the PSPACE-hardness proof for the very similar thorough refinement in Paper E we conjecture that this refinement is indeed PSPACE-hard.

5 Thesis Summary

Paper A: Modeling Software Product Lines Using Color-blind Transition Systems

Families of embedded discrete finite state programs are modeled using input-enabled alternating transition systems. One model describes all functionality, while each variant is defined by an environment, describing its possible uses. The environments show both the inputs that a system can receive and indicate which of the system's responses are relevant for the environment. The latter trait, called color-blindness, creates new possibilities for system transformations in the specialization process. We demonstrate the use of the framework by applying it to two classes of realistic design languages. An example of a product line of alarm clocks is used throughout the article.

Contributions

- Introduces the concept of color-blindness; an environments inability to see the difference between certain output.
- Describes a framework for product lines that can be instantiated for different design languages.
- Describes how specialized models can be automatically generated from one family-model and several color-blind environment models.
- Applies the product line framework to two different, realistic design languages.

Publication History

A short abstract on the research was presented at the 16th *Nordic Workshop on Programming Theory* (NWPT'04). An earlier version of this paper [LLW05] was presented at the 8th international conference on *Formal Approaches to Software Engineering* (FASE'05) and published in LNCS volume 3442 [Cer05]. The present version was published in a special issue of the *International Journal on Software Tools for Technology Transfer* (STTT) devoted to FASE'04/05. [LNW07b]

Paper B: Interface Input/Output Automata

Building on the theory of interface automata by de Alfaro and Henzinger we design an interface language for Lynch's I/O automata, a popular formalism used in the development of distributed asynchronous systems, not addressed by previous interface research. We introduce an explicit separation of assumptions from guarantees not yet seen in other behavioral interface theories. Moreover we derive the composition operator systematically and formally, guaranteeing that the resulting compositions are always the weakest in the sense of assumptions, and the strongest in the sense of guarantees. We also present a method for solving systems of relativized behavioral inequalities as used in our setup and draw a formal correspondence between our work and interface automata.

Contributions

- An interface theory based on I/O automata.
- Introduces an explicit split of assumptions from guarantees in the specification of interfaces.
- A formally derived composition operator for interfaces, guaranteeing that the resulting compositions are always the weakest in the sense of assumptions, and the strongest in the sense of guarantees.
- Presents a method for solving systems of relativized behavioral inequalities.
- A formal correspondence between our work and Interface Automata [AH01].

Publication History

The first version of this research was presented at the first workshop on *Foundations of Interface Technologies* (FIT 2005) held in conjunction with CONCUR 2005. The current version of the work [LNW06b] was presented at the 14th International Symposium on *Formal Methods* (FM 2006) and published in LNCS volume 4085 [MNS06]. An extended version including all proofs and extra material was published as a technical report [LNW06a].

Paper C: Modal I/O Automata for Interface and Product Line Theories

Alfaro and Henzinger use alternating simulation in a two player game as a refinement for interface automata [AH01]. We show that interface automata correspond to a subset of modal transition systems of Larsen and Thomsen [LT88], on which alternating simulation coincides with modal refinement. As a consequence a more expressive interface theory may be built, by a simple generalization from interface automata to modal automata. We define modal I/O automata, an extension of interface automata with modality. Our interface theory that follows can express liveness properties, disallowing trivial implementations of interfaces, a problem that exists for theories build around simulation preorders. In order to further exemplify the usefulness of modal I/O automata, we construct a behavioral variability theory for product line development.

Contributions

- Shows that Interface Automata [AH01] corresponds to a subset of modal transition systems of Larsen and Thomsen [LT88]. On this subset alternating simulation coincides with modal refinement
- Formally defines modal I/O automata, an extension of interface automata with modality.
- Describes an interface theory which can express liveness properties, disallowing trivial implementations of interfaces.
- Describes a behavioral variability theory for product line development.

Publication History

The paper [LNW07a] was presented at the 16th *European Symposium on Programming* (ESOP 2007) and published in LNCS volume 4421 [Nic07].

Paper D: On Modal Refinement and Consistency

Almost 20 years after the original conception, we revisit several fundamental question about modal transition systems. First, we demonstrate the incompleteness of the standard modal refinement using a counterexample due to Hüttel. Deciding any refinement, complete with respect to the standard notions of implementation, is shown to be computationally hard (co-NP hard). Second, we consider four forms of consistency (existence of implementations) for modal specifications. We characterize each operationally, giving algorithms for deciding, and for synthesizing implementations, together with their complexities.

Contributions

- Demonstrates the incompleteness of the standard modal refinement using a counterexample due to Hüttel.
- Deciding any refinement, complete with respect to the standard notions of implementation, is shown to be computationally hard (co-NP hard).
- Operational characterizations of four different forms of consistency.
- Algorithms for synthesizing implementations for all four consistencies.

Publication History

The paper [LNW07c] was presented at the 18th International *Conference on Concurrency Theory* (CONCUR 2007) and published in LNCS volume 4703 [CV07].

Paper E: Complexity of Decision Problems for Mixed and Modal Specifications

We consider decision problems for modal and mixed transition systems used as specifications: the *common implementation* problem (whether a set of specifications has a common implementation), the *consistency* problem (whether a single specification has an implementation), and the *thorough refinement* problem (whether all implementations of one specification are also implementations of another one). *Common implementation* and *thorough refinement* are shown to be PSPACE-hard for modal, and so also for mixed, specifications. *Consistency* is PSPACE-hard for mixed, while trivial for modal specifications. We also supply upper bounds suggesting strong links between these problems.

Contributions

- Proves PSPACE-hardness for both mixed and modal specification for *common implementation* and *thorough refinement*.
- Proves PSPACE-hardness for *consistency* of mixed specifications.
- Shows an upper bound of EXPTIME for *common implementation*, *consistency* and *thorough refinement*.
- Establishes a reduction from *common implementation* of mixed and modal specification to *consistency* of mixed specifications.
- Establishes a reduction from *consistency* of mixed specifications to *thorough reduction* of mixed specifications.

Publication History

The paper [AHL+08] was presented at the Eleventh International Conference on *Foundations of Software Science and Computation Structures* (FoSSaCS 2008) and published in LNCS volume 4962.

Paper A

Modeling Software Product Lines Using Color-blind Transition Systems

Kim G. Larsen, Ulrik Nyman
Department of Computer Science,
Aalborg University, Denmark

Andrzej Wąsowski
Computational Logic and Algorithms Group,
IT University of Copenhagen, Denmark

Abstract

Families of embedded discrete finite state programs are modeled using input-enabled alternating transition systems. One model describes all functionality, while each variant is defined by an environment, describing its possible uses. The environments show both the inputs that a system can receive and indicate which of the system's responses are relevant for the environment. The latter trait, called color-blindness, creates new possibilities for system transformations in the specialization process. We demonstrate the use of the framework by applying it to two classes of realistic design languages. An example of a product line of alarm clocks is used throughout the article.

Keywords: Product Lines, Embedded Software, Labeled Transition Systems, Modeling, Relativized Simulation

1 Introduction

Modern software becomes increasingly customizable. Embedded devices are often produced in multiple variants, each needing different software. Our long-term goal is to provide a theoretical foundation, tools, and a methodology for maintenance of a family of embedded software products with similar but varying degree of functionality. Such a family of products is usually known as a product line, and the process of maintaining the software is known as product line management [CN01, Gom01]. The present work focuses on a framework for modeling software product lines and specifying correctness of transformations used in automatic derivation of family members.

In design of embedded software the final code size is often essential for the financial success of the product. Even a small reduction in code size may allow reductions in the production cost, if it makes it possible to use cheaper hardware. Our communications with vendors confirm that many electronic systems are produced in such large quantities that even a small reduction for each device can bring enormous savings.

Manual implementation of the different versions of software for different versions of the product can keep the final code sizes close to optimal, while driving up the software development cost significantly. The general idea of product line management is to avoid explicit maintenance of multiple versions of the software, and use methods for design and development of the entire product family as a whole.

We propose to use a single general model as a description of all available functionality in a product family. Such a family may evolve over time, and so can the general model, but we do not consider this here. A set of hierarchically organized specifications describes the different environments in which each version of the embedded software will operate. Pragmatically speaking environments are descriptions of all possible uses for a given variant of the product. A simpler system variant usually allows fewer sensor inputs and fewer actuator outputs, which corresponds to environments *not providing* the inputs, or *not caring* about specific responses received. We will say that the implementation of the system has been specialized correctly to a given variant if the restricting environment cannot see the difference between the original and specialized versions of the program.

Fig. A.1 gives an overview of the setup. The designer needs to create the model containing the complete set of functionality (top-left state diagram) and the environment specifications (top row). In the depicted setup the general model is specialized to a given environment in the model language before being compiled into the target language. The model obtained after the specialization should behave identically to the general model as long as the environment behaves according to its specification. In pragmatic terms this means that if the environment models the hardware and the system is specialized with respect to this model, it should behave as expected as long as it is run on the hardware respecting the

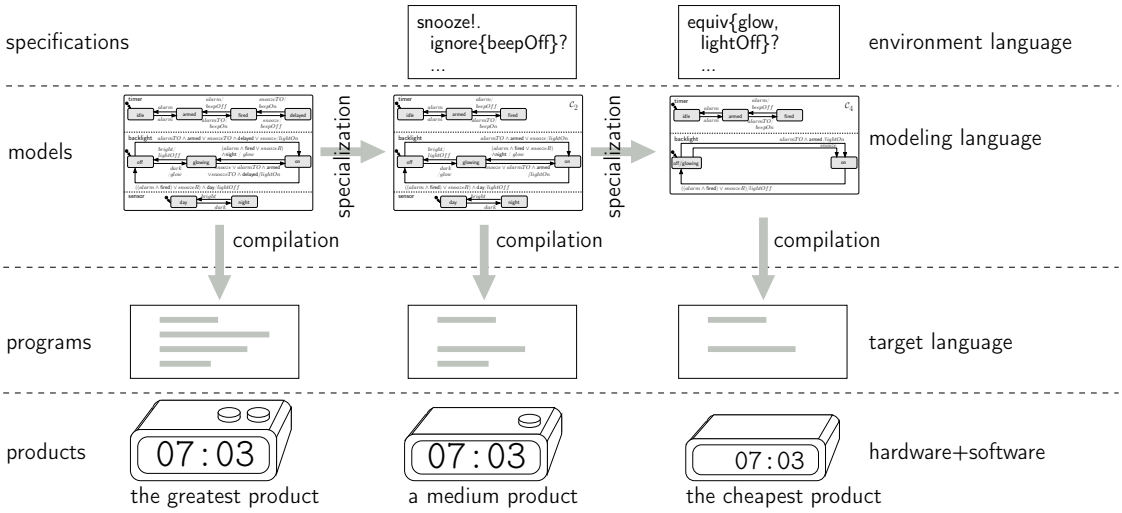


Figure A.1: Overview of a product line of alarm clocks. The top left model is the general model specified by a developer. User also provides usage descriptions in the form of environments (on the very top).

assumptions.

A compelling example of how different two versions of the same product can be, can be found on Fig. A.2, which shows the front of two different coffee vending machines by Wittenborg. The simple machine, FB 55, has no coin collection unit or choice of additives, such as sugar or milk. Such a simple coffee machine is ideal in circumstances where the coffee is pre-paid, and where the simplicity is important because people are in a hurry. Indeed many even simpler Wittenborg's machines have recently been seen at open areas of Frankfurt and Munich airports. The more complex machine, ES 5100, is more appropriate for an environment where people use the same machine every day, such as at work (one can be seen in CISS, where two of the authors are employed). The control software for these two machines could have been constructed by creating one model containing the comprehensive functionality of the most complex coffee machine and then creating environments for simpler coffee machines. The environments could state that the user would never use certain buttons, in that these buttons would not be physically present on the machine.

Throughout the article we will use an alarm clock example. In order to produce several different versions of alarm clocks we only need to design a general model with all the functionality (Fig. A.3) and specify the environments in which the other versions of the alarm clocks are expected to function. Fig. A.7 shows a very restricted version of the alarm clock, with much less functionality. This alarm clock can be derived from the original model given a description of an environment.

Figure A.2: Two variants of Wittenborg's coffee machine. The simple FB 55 (left) has few buttons and no coin collection unit. The advanced ES 5100 (right) provides the user with the choice of different coffee specialties and additives.

Our environments not only restrict possible input stimuli, but also exhibit inabilities in distinguishing output stimuli. Some outputs are indistinguishable for a given environment in the same way as a color-blind person cannot distinguish some colors. In the case of Fig. A.7 the environment cannot, among other things, see the difference between the light in the clock glowing or being completely off. Inability of a given environment to distinguish certain outputs is dependent on the state of that environment. Thus an environment can specify that a user, after pressing one button, will not distinguish between two different outputs, but will distinguish between these two outputs again after pressing another button, e.g. a reset button.

The paper proceeds as follows. Section 2 introduces state/event systems and motivates our work using the alarm clock example. I/O alternating transition systems are presented in section 3, while the concept of color-blindness is introduced in section 4. Composition operators for environments are discussed in section 5. Section 6 focuses on adaptation to realistic design languages. The main alarm clock example is completed in section 7. In section 8 we briefly touch on available techniques that can be used in specializing code for product variants. Sections 7–10 refer the related work and conclude.

2 State/Event Systems

Let *Event* and *Action* be finite sets of environment stimuli and system outputs respectively. A *state/event machine* $M_i = (S_i, s_i^0, T_i)$ is a triple comprising a set of local states S_i, the initial state $s_i^0 \in S_i$ and a set of syntactic transitions T_i. A *state/event system* consists of n machines $\mathcal{M} = \{M_1, \ldots, M_n\}$ with mutually disjoint sets of states. A global state of the system is a tuple of local states: $State = S_1 \times S_2 \times \cdots \times S_n$. Transitions in $T_i \subseteq S_i \times Event \times Guard \times Action \times S_i$ describe reactions undertaken by M_i in reply to a given event, in a given local and global state. Global states are described by transition guards: simple Boolean expressions over activity of states, which can be evaluated in any given global state, giving rise to a natural satisfaction relation $\vDash \subseteq State \times Guard$.

State/event systems are *input-enabled*: the local transition relation includes not only the syntactical transitions but also self loops for all configurations for which reactions are not specified. We write $s \xrightarrow{e \quad o}_i s_i'$, meaning that the reaction of machine M_i to arrival of event e in global state s is, to change the local state to s_i' and generate the set of actions o:

$$\begin{array}{ll} s \xrightarrow{e \quad \{a\}}_i s_i' & \text{iff } \exists g.\ (\pi_i(s), e, g, a, s_i') \in T_i \wedge s \vDash g \\ s \xrightarrow{e \quad \emptyset}_i \pi_i(s) & \text{otherwise,} \end{array}$$

where $\pi_i(s)$ denotes the i'th projection of s. The global transition relation

$$T \subseteq State \times Event \times \mathcal{P}(Action) \times State$$

subsumes all local reactions

$$s \xrightarrow{e \quad o} s' \Leftrightarrow_{def} \forall i. s \xrightarrow{e \quad o_i}_i \pi_i(s') \text{ where } o = o_1 \cup \ldots \cup o_n.$$

Fig. A.3 depicts a state/event model $\mathcal{C}_0$ of an alarm clock. The essentials of the alarm clock are handled by the *timer* machine. If the *timer* is in the armed state and the hardware sends an alarm time-out event (*alarmTO*) then the beeper is turned on. The actual timers are not part of the modeled system, and thus the environment sends a time-out event to the system with a certain delay after a hardware timer has been activated. The user can postpone the alarm by pressing the snooze button (event *snooze*), which allows him to continue sleeping until the snooze timer times out (*snoozeTO*). Releasing the button sends a *snoozeR* event to the model. The *backlight* machine controls the built-in lamps. Only a faint light is displayed in the glowing state, such that the display can be read in the dark. The full light is on while the alarm is beeping or the snooze button is being pressed. The *sensor* machine models the current external light level. Events *dark* and *bright* are generated by the sensor driver whenever the light level around the clock changes above or below some threshold.

We would like to support automatic derivation of model variants for discrete control systems like the alarm clock. One such variant $\mathcal{C}_4$, which is a very limited version of the alarm clock is depicted on Fig. A.7. This variant does not have any snooze functionality, because the environment guarantees that it will never provide the input *snooze*. Thus it is as if there were no snooze button on the alarm clock. This environment also specifies that it cannot see the difference between the backlight glowing and being completely off. This implies that the hardware of the clock would only need one lightbulb, for the full light setting, instead of two lightbulbs for two different light levels. The glowing and off states can now be combined into one state and the *sensor* component is no longer necessary.

3 I/O Alternating Transition Systems

The reactive synchronous paradigm seems to be predominant in development of embedded software. The state/event systems [LNAH+01, IAR] of the previous section are just an example chosen from a multitude of available formalisms, like Esterel [Ber00], statecharts [Har87], or Java Card [Sun]. A common assumption about these systems is that they react to any input event at any time and each reaction occurs infinitely fast, so that the system is always able to observe the arrival of the next event. Such semantics is conveniently captured by *I/O-alternating transition systems*:

Definition 1 *An I/O-alternating transition system, or IOATS, is a tuple* $(In, Out, Gen, Obs, \xrightarrow{!}, \xrightarrow{?}, s^0)$*, where In and Out are sets of inputs and outputs, Gen and Obs are finite sets of generators and observers,* $\xrightarrow{!} \subseteq Gen \times Out \times Obs$ *is a generation relation,* $\xrightarrow{?} \subseteq Obs \times In \times Gen$ *is an observation relation, and* $s^0 \in Gen \cup Obs$ *is the initial state.*

We have distinguished two transition relations: $\xrightarrow{!}$ is a generation relation advancing from a generator to an observer, while $\xrightarrow{?}$ is an observation relation advancing from an observer to a generator. This alternation is inherent to the way synchronous systems operate. We write $S \xrightarrow{o!} s$, instead of $(S, o, s) \in \xrightarrow{!}$ and $s \xrightarrow{i?} S$ instead of $(s, i, S) \in \xrightarrow{?}$. Small letters are used for observers and capital letters for generators. In addition observers are required to be input-enabled:

$$\forall s \in Obs.\ \forall i \in In.\ \exists S, o, s'.\ s \xrightarrow{i?} S \wedge S \xrightarrow{o!} s' \tag{A.1}$$

With these assumptions we can propose a simulation based refinement relation:

Definition 2 *Let* $\mathcal{S}_1 = (In, Out, Gen_1, Obs_1, \xrightarrow{!}_1, \xrightarrow{?}_1, s_1^0)$ *and* $\mathcal{S}_2 = (In, Out, Gen_2, Obs_2, \xrightarrow{!}_2, \xrightarrow{?}_2, s_2^0)$ *be IOATSs. A binary relation* $R \in Obs_1 \times Obs_2$ *constitutes a simulation on observers of* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *iff* $(s_1, s_2) \in R$ *implies that:*

$$\textit{whenever } s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o!} s_1' \textit{ then also } s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o!} s_2' \textit{ and } (s_1', s_2') \in R\ .$$

Let R be the largest of such relations ordered by inclusion. An observer s_2 simulates an observer s_1, written $s_1 \leqslant s_2$, iff $(s_1, s_2) \in R$. Finally $\mathcal{S}_2$ simulates $\mathcal{S}_1$, written $\mathcal{S}_1 \leqslant \mathcal{S}_2$, iff $s_1^0 \leqslant s_2^0$.

We distinguish the actual systems from the environments, in which they operate. Environments are free in choice of inputs, while systems independently determine the outputs. A system $\mathcal{S} = (In_\mathcal{S}, Out_\mathcal{S}, Gen_\mathcal{S}, Obs_\mathcal{S}, \xrightarrow{!}_\mathcal{S}, \xrightarrow{?}_\mathcal{S}, s_\mathcal{S})$ operates embedded in some environment $\mathcal{E} = (In_\mathcal{E}, Out_\mathcal{E}, Gen_\mathcal{E}, Obs_\mathcal{E}, \xrightarrow{!}_\mathcal{E}, \xrightarrow{?}_\mathcal{E}, s_\mathcal{E})$. Systems always begin execution in an observer state, so $s_\mathcal{S} \in Obs_\mathcal{S}$. Environments always begin execution in a generator state, so $s_\mathcal{E} \in Gen_\mathcal{E}$. System $\mathcal{S}$ is *compatible* with the environment $\mathcal{E}$ if $In_\mathcal{S} = Out_\mathcal{E}$ and $Out_\mathcal{S} = In_\mathcal{E}$. Composition of a system $\mathcal{S}$ with a compatible environment $\mathcal{E}$ is defined in the usual way, by synchronization on identical labels (and complementary transition types). The initial observer of the system is composed with the initial generator of the environment. Due to the compatibility requirement and input-enabledness of observers, the closed system is able to advance for any input that can be generated by the environment. For a closed system it is known, which of its states cannot be exercised by the environment. A given environment may not be able to distinguish two systems from each other, even though they are not identical. We capture this with a notion of *relativized simulation*:

Definition 3 *Consider three IOATSs: an environment $\mathcal{E} = (Out, In, Gen, Obs, \xrightarrow{!}, \xrightarrow{?}, E^0)$ and two systems: $\mathcal{S}_1 = (In, Out, Gen_1, Obs_1, \xrightarrow{!}_1, \xrightarrow{?}_1, s_1^0)$ and $\mathcal{S}_2 = (In, Out, Gen_2, Obs_2, \xrightarrow{!}_2, \xrightarrow{?}_2, s_2^0)$. A Gen-indexed family of binary relations $R : Gen \to \mathcal{P}(Obs_1 \times Obs_2)$ is a relativized simulation iff $(s_1, s_2) \in R_E$ implies that:*

$$\begin{array}{l} \textit{whenever } E \xrightarrow{i!} e \wedge e \xrightarrow{o?} E' \\ \quad \textit{then whenever } s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o!} s_1' \\ \quad\quad \textit{then also } s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o!} s_2' \textit{ and } (s_1', s_2') \in R_{E'}. \end{array}$$

Let R be the largest of such families ordered by component-wise inclusion. We say that an observer s_2 simulates an observer s_1 in the generator E, written $E \models s_1 \leqslant s_2$, iff $(s_1, s_2) \in R_E$. The system $\mathcal{S}_2$ simulates $\mathcal{S}_1$ in the context of $\mathcal{E}$, written $\mathcal{E} \models \mathcal{S}_1 \leqslant \mathcal{S}_2$, iff $E^0 \models s_1^0 \leqslant s_2^0$.

The choice of simulation as the preorder underlying our methodology is somewhat arbitrary. Most other behavioral preorders of the linear-time/branching-time hierarchy of van Glabbeek [vG90] would be adequate, such as testing preorder, ²⁄₃ bisimulation (ready simulation), language inclusion, ioco [Tre96] and bisimulation. What is important is that the particular preorder preserves properties of interest and that the preorder may be relativized with respect to environmental restrictions.

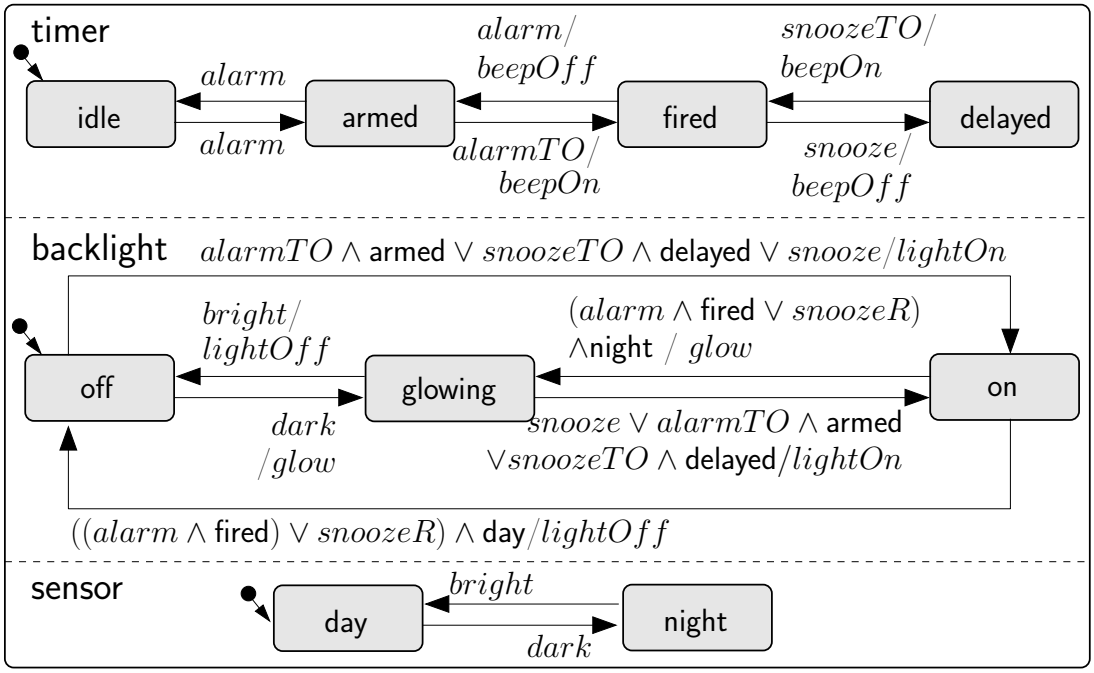


Figure A.3: The initial model, $\mathcal{C}_0$, of the alarm clock

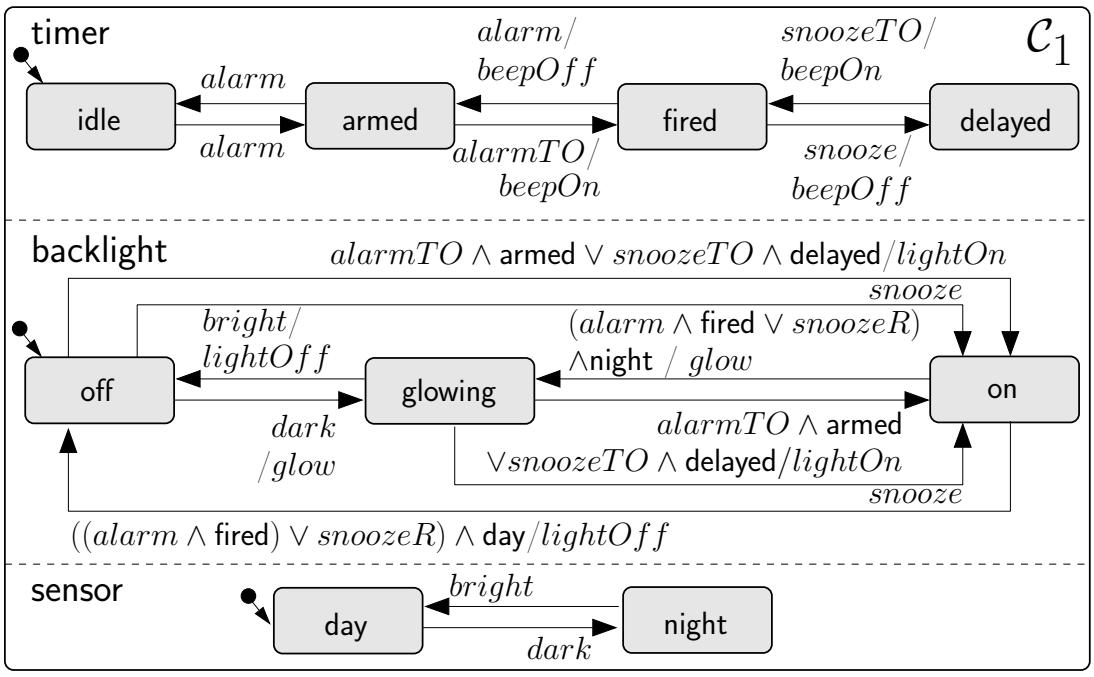


Figure A.4: A specialized model, $\mathcal{C}_1$, of an alarm clock.

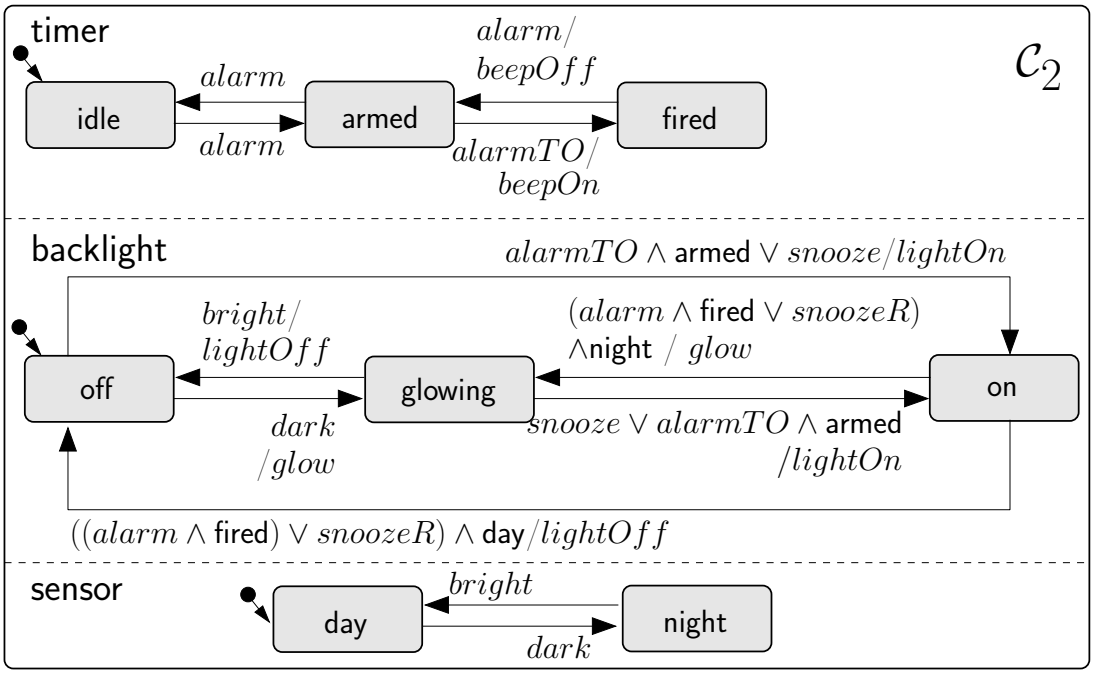


Figure A.5: A specialized model, $\mathcal{C}_2$, of an alarm clock.

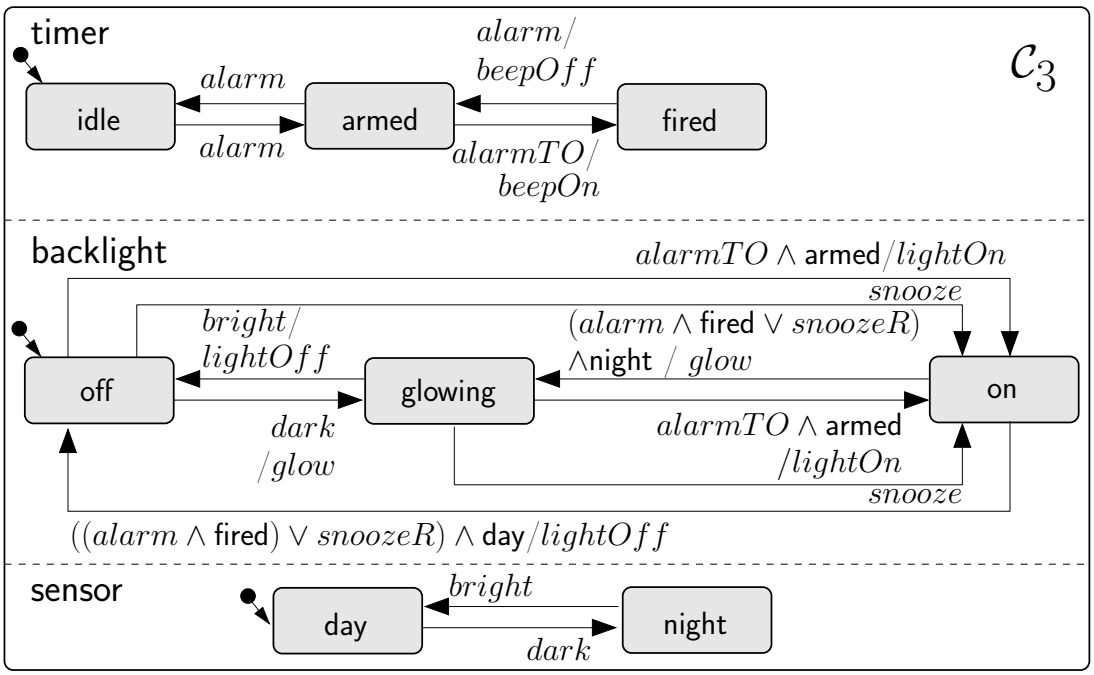


Figure A.6: A specialized model, $\mathcal{C}_3$, of an alarm clock.

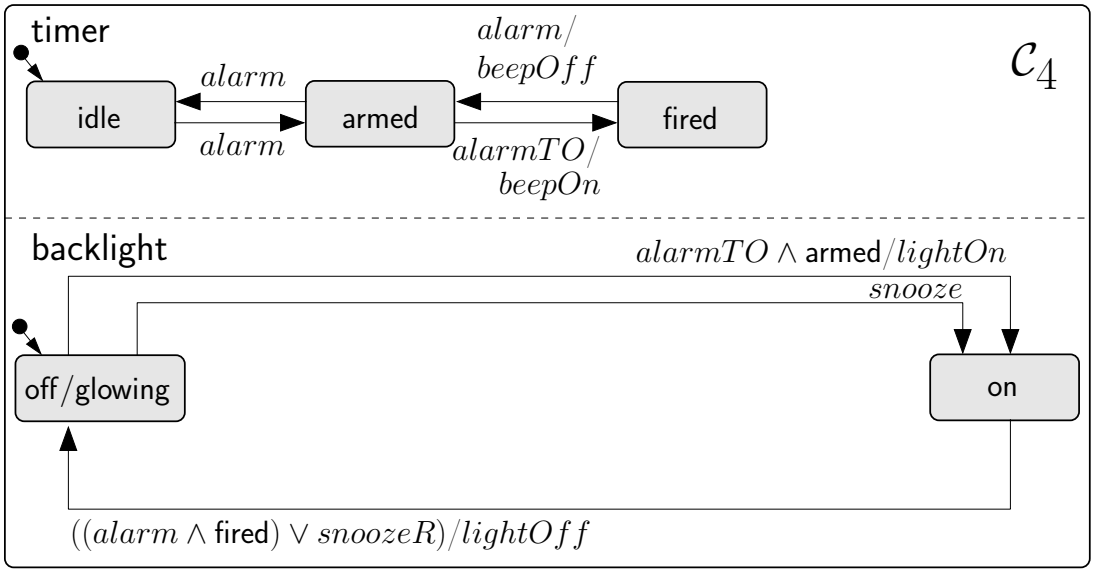


Figure A.7: A specialized model, $\mathcal{C}_4$, of an alarm clock

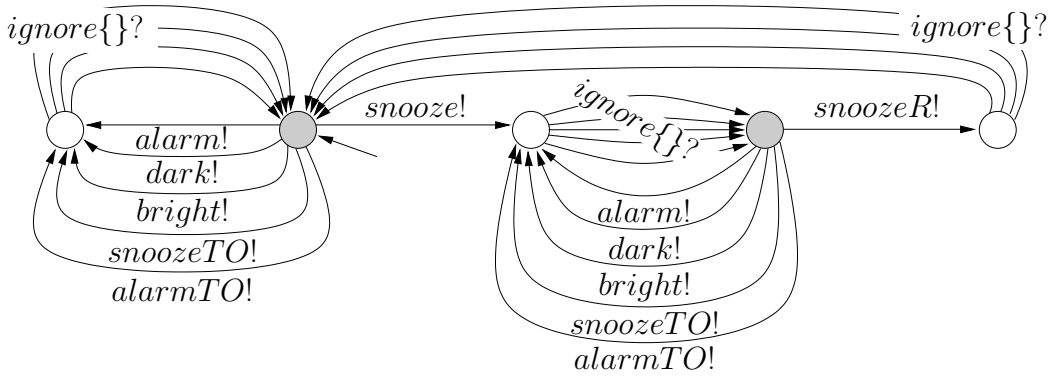


Figure A.8: Environment *Interleave snooze snoozeR*.

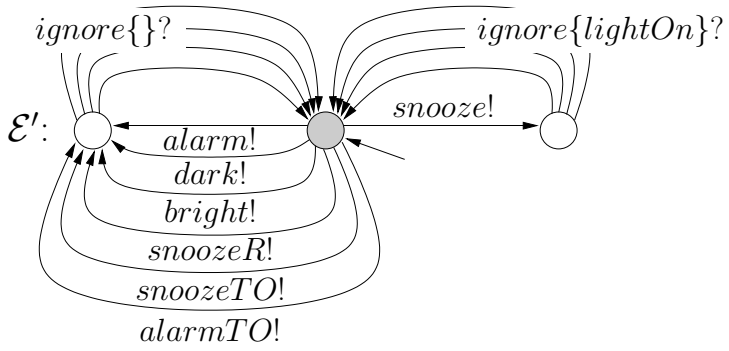


Figure A.9: Environment $\mathcal{E}'$ ignoring the *lightOn* output produced in reaction to the *snooze* button.

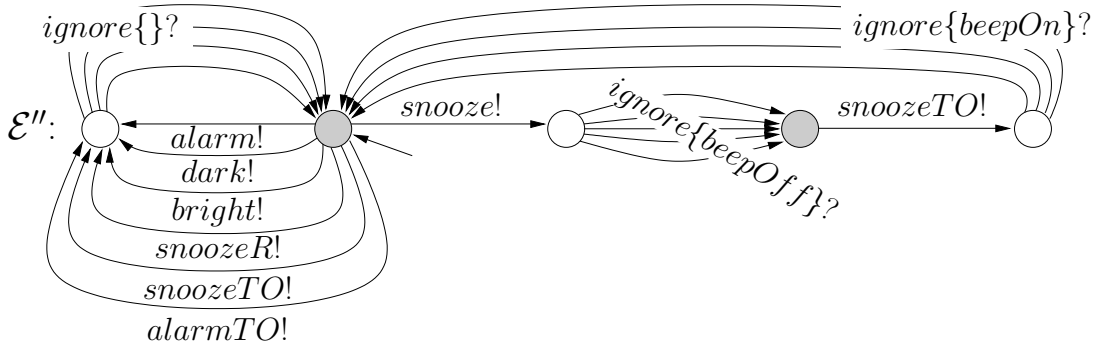


Figure A.10: Environment $\mathcal{E}''$ ignoring the snooze function of the clock.

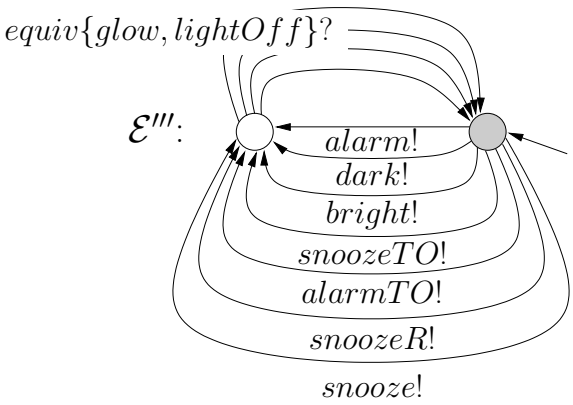


Figure A.11: Environment $\mathcal{E}'''$ *Equiv glow lightOff*.

4 Color-blind I/O-alternating Transition Systems

In the previous section we have stated that two systems are in a refinement relation with respect to a certain context if this context cannot activate their incompatible parts. However, in industrial development, it often happens that the environment cannot distinguish two systems, not because it makes incompatible parts unreachable, but because its ability to distinguish *different outputs* might be limited depending on its actual state. A variant of the alarm clock may have only one lightbulb installed, which should be lit whenever the backlight is on or glowing. The environment, being a model of the hardware in this case, will treat the *glow* and *lightOn* outputs as identical, allowing for optimizations when generating code for this specific type of hardware.[1] For this particular example, the distinguishing capability of the environment is clearly static and hence the specification of code optimization is realizable using simple process algebraic operations such as relabelling and hiding. However, environmental restrictions can be dynamically changing. This is the case for the environment leading to the specialized model $\mathcal{C}_1$ (Fig. A.4). Here the environment only becomes blind for the *lightOn* action after the production of the *snooze* event. To give a proper treatment of such situations we relax the equivalence of labels in relativized simulation and label observation transitions of environments with sets of inputs called *observation classes*. Such transitions can be taken in the presence of any of the inputs in their observation class.

Definition 4 *A* color-blind *IOATS is a tuple* $\mathcal{E} = (In, Out, Gen, Obs, \xrightarrow{!}, \xrightarrow{?}, E^0)$*, where In and Out are sets of inputs and outputs, Gen and Obs are finite sets of generators and color-blind observers,* $\xrightarrow{!} \subseteq Gen \times Out \times Obs$ *is a generation relation,* $\xrightarrow{?} \subseteq Obs \times \mathcal{P}(In) \times Gen$ *is a color-blind observation relation, and* $E^0 \in Gen$ *is an initial state.*

A color-blind environment $\mathcal{E} = (In_\mathcal{E}, Out_\mathcal{E}, Gen_\mathcal{E}, Obs_\mathcal{E}, \xrightarrow{!}_\mathcal{E}, \xrightarrow{?}_\mathcal{E}, E)$ and a usual IOATS $\mathcal{S} = (In_\mathcal{S}, Out_\mathcal{S}, Gen_\mathcal{S}, Obs_\mathcal{S}, \xrightarrow{!}_\mathcal{S}, \xrightarrow{?}_\mathcal{S}, s)$ are compatible if their signatures match: $In_\mathcal{E} = Out_\mathcal{S} \wedge Out_\mathcal{E} = In_\mathcal{S}$. Since we only consider compatible systems and environments, we fix the meaning of the input and output, choosing the system's perspective. We denote the set of inputs of the system by *In* (which is also the set of outputs of the environment). Similarly *Out* is the set of outputs of the system (but the set of inputs for the environment). A single input will be denoted by i, single output by o, and classes of outputs by capital O. We still write $E \xrightarrow{i!} e$ instead of $(E, i, e) \in \xrightarrow{!}$ and $e \xrightarrow{O?} E$ instead of $(e, O, E) \in \xrightarrow{?}$.

We require that the observers in color-blind IOATS are deterministic and input enabled, so that the observation classes on the transitions outgoing from a

[1] Note that this is an alternative to the version presented in Fig. A.7, where *glow* and *lightOff* are considered identical.

single state form a partitioning of the inputs into equivalence classes. Formally:

$$\begin{aligned}
&\forall e \in Obs_{\mathcal{E}}.\forall O_1, O_2 \subseteq Out.\forall E_1, E_2 \in Gen_{\mathcal{E}}.\\
&\quad e\xrightarrow{O_1?}E_1 \wedge e\xrightarrow{O_2?}E_2\\
&\quad\Rightarrow O_1 \cap O_2 = \emptyset \vee (O_1 = O_2 \wedge E_1 = E_2)\\
&\forall e \in Obs_{\mathcal{E}}.\forall o \in Out.\exists O \subseteq Out.\exists E \in Gen_{\mathcal{E}}.\\
&\quad e\xrightarrow{O?}E \wedge o \in O.
\end{aligned} \tag{A.2}$$

The generation relation should also be deterministic:

$$\forall E \in Gen_{\mathcal{E}}.\ \forall i \in In.\forall e_1, e_2 \in Obs_{\mathcal{E}}.E\xrightarrow{i!}e_1 \wedge E\xrightarrow{i!}e_2 \Rightarrow e_1 = e_2\ . \tag{A.3}$$

Note that determinism in this sense does not limit the freedom of the environment in choosing inputs, but means that each input choice uniquely determines the target state.

Consider a blind environment $\mathcal{B}$ with two states, a generator $\mathbf{B}$ and an observer $\mathbf{b}$. Intuitively $\mathcal{B}$ can execute all parts of the system, but does not care about the responses it gets:

$$\forall i \in In.\ \mathbf{B}\xrightarrow{i!}\mathbf{b} \text{ and } \mathbf{b}\xrightarrow{Out?}\mathbf{B}\ .$$

Dually, a perfect vision environment $\mathcal{V}$ observes all the outputs:

$$\forall i \in In.\mathbf{V}\xrightarrow{i!}\mathbf{v} \text{ and } \forall o \in Out.\mathbf{v}\xrightarrow{\{o\}?}\mathbf{V}\ .$$

A compatible environment–system pair forms a closed system, advancing in lock-steps. The generation transition of the system, synchronizes with the observation transition of the environment, whenever the output produced falls into the right observation class. We enrich our previous definition of relativized simulation to accommodate this new synchronization principle:

Definition 5 *Let $\mathcal{E} = (Out, In, Gen, Obs, \xrightarrow{!}, \xrightarrow{?}, E^0)$ be a color-blind environment IOATS and $\mathcal{S}_1 = (In, Out, Gen_1, Obs_1, \xrightarrow{!}_1, \xrightarrow{?}_1, s_1^0)$, $\mathcal{S}_2 = (In, Out, Gen_2, Obs_2, \xrightarrow{!}_2, \xrightarrow{?}_2, s_2^0)$ be two system IOATSs. A Gen-indexed family of relations $R\colon Gen\to\mathcal{P}(Obs_1\times Obs_2)$ is a relativized simulation iff $(s_1, s_2)\in R_E$ implies that:*

$$\begin{aligned}
&\textit{whenever } E\xrightarrow{i!}e \wedge e\xrightarrow{O?}E'\\
&\quad\textit{then whenever } s_1\xrightarrow{i?}S_1 \wedge S_1\xrightarrow{o_1!}s_1' \wedge o_1 \in O\\
&\qquad\textit{then also } s_2\xrightarrow{i?}S_2 \wedge S_2\xrightarrow{o_2!}s_2' \wedge o_2 \in O\\
&\qquad\quad\textit{and } (s_1', s_2') \in R_{E'}.
\end{aligned}$$

Let R be the largest of such families ordered by component-wise inclusion. An observer s_2 simulates an observer s_1 in the context of generator E, written $E \models s_1 \leqslant s_2$, iff $(s_1, s_2) \in R_E$. An IOATS $\mathcal{S}_2$ simulates another IOATS $\mathcal{S}_1$ in the context of a compatible color-blind IOATS $\mathcal{E}$, written $\mathcal{E} \models \mathcal{S}_1 \leqslant \mathcal{S}_2$, iff $E^0 \models s_1^0 \leqslant s_2^0$. Finally $\mathcal{S}_1$ is equivalent to $\mathcal{S}_2$ in the context of $\mathcal{E}$, written $\mathcal{S}_1 \leqslant_{\mathcal{E}} \mathcal{S}_2$, iff $\mathcal{E} \models \mathcal{S}_1 \leqslant \mathcal{S}_2$ and $\mathcal{E} \models \mathcal{S}_2 \leqslant \mathcal{S}_1$.

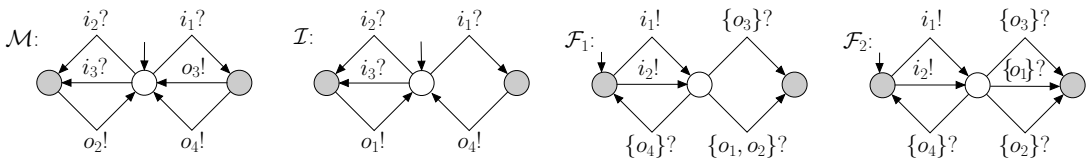


Figure A.12: Systems $\mathcal{M}$ and $\mathcal{I}$ and compatible environments $\mathcal{F}_1$, $\mathcal{F}_2$

Let $\mathcal{S}_1$, $\mathcal{S}_2$ be IOATSs and $\mathcal{E}$ be a color-blind IOATS compatible with them, as in the above definition. Let $\mathbb{R}$ be an endofunction on a *Gen*-indexed family of binary relations:

$$\begin{aligned}\mathbb{R}(R) = \lambda E.\{(s_1, s_2) \mid \forall i, e, O, E'.\ \forall i, S_1, o_1, s_1'.\ \exists S_2, o_2.\\ E \xrightarrow{i!} e \wedge e \xrightarrow{O?} E' \wedge s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o_1!} s_1' \wedge o_1 \in O\\ \text{implies } s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o_2!} s_2' \wedge o_2 \in O \wedge (s_1', s_2') \in R_{E'}\}\ .\end{aligned}$$

Proposition 1 *A Gen-indexed family of relations R constitutes a relativized simulation with respect to a color-blind IOATS iff $R \subseteq \mathbb{R}(R)$ (inclusion interpreted component-wise).*

Proof 1 *$\mathbb{R}$ is a monotonic endofunction on the complete lattice of Gen-indexed famillies of binary relations over $Obs_1 \times Obs_2$ ordered by inclusion. By Tarski's theorem [Tar55] $\mathbb{R}$ has the greatest fixpoint $\bigcap_{j=0}^{\infty} \mathbb{R}^j(\lambda E.\, Obs_{\mathcal{S}_1} \times Obs_{\mathcal{S}_2})$, equal to the relativized simulation of Def. 5. Since the fixpoint contains all relativized simulations, we can use a classic proof technique: to show that one IOATS simulates another in an environment, find any relativized simulation relating them.* □

Even though we have initially postulated that typical execution contexts do not exercise all possible traces of the system, we shall now require that environments can always produce any of the inputs in *In*. This requirement surprisingly does not defeat our initial goal. We can direct all transitions producing impossible inputs to the observer **b** and embed the blind environment $\mathcal{B}$ with a suitable signature in every environment. Instead of specifying that the environment cannot produce i, we state that i can be produced, but the subsequent system behavior is irrelevant. Proposition 2 justifies this formally:

Proposition 2 *For any two observers s_1, s_2 from IOATSs $\mathcal{S}_1$, $\mathcal{S}_2$ with identical signatures: $\mathbf{B} \models s_1 \leqslant s_2$ (where $\mathbf{B} \in \mathcal{B}$ such that $\mathcal{B}$ closes $\mathcal{S}_1$ and $\mathcal{S}_2$).*

Fig. A.12 presents two systems and two compatible color-blind environments. Environment transitions from generators to the blind observer **b** have been omitted. There is one such transition for each input–generator pair, for which the transition is not drawn. Observe that the system $\mathcal{M}$ simulates $\mathcal{I}$ in the environment $\mathcal{F}_1$ (written $\mathcal{F}_1 \models \mathcal{I} \leqslant \mathcal{M}$) not due to the fact that $\mathcal{F}_1$ is not able to exercise the differing parts of the two systems, but because $\mathcal{F}_1$ cannot distinguish between

the outputs (o_1, o_2) produced by $\mathcal{I}$ and $\mathcal{M}$. The $\mathcal{F}_2$ environment distinguishes $\mathcal{I}$ and $\mathcal{M}$, by observing the outputs o_1 and o_2 with two separate transitions.

Relativized simulation is a weaker notion than usual simulation and the perfect vision environment $\mathcal{V}$ is the most discriminating environment:

Proposition 3 *For any two systems $\mathcal{S}_1$, $\mathcal{S}_2$ and for any compatible color-blind environment $\mathcal{E}$ it holds that $\mathcal{S}_1 \leqslant \mathcal{S}_2 \implies \mathcal{E} \models \mathcal{S}_1 \leqslant \mathcal{S}_2$ and $\mathcal{S}_1 \leqslant \mathcal{S}_2 \iff \mathcal{V} \models \mathcal{S}_1 \leqslant \mathcal{S}_2$.*

With the above propositions we have hinted at the notion of *discrimination*—the ability of environment to distinguish systems from each other:

Definition 6 *A color-blind IOATS$\mathcal{F}$ is more discriminating than $\mathcal{E}$, written $\mathcal{E} \sqsubseteq \mathcal{F}$, iff $\mathcal{F}$ distinguishes more processes: $\mathcal{E} \sqsubseteq \mathcal{F}$ iff $\forall \mathcal{S}_1, \mathcal{S}_2 . \mathcal{F} \models \mathcal{S}_1 \leqslant \mathcal{S}_2 \Rightarrow \mathcal{E} \models \mathcal{S}_1 \leqslant \mathcal{S}_2$.*

The blind environment $\mathcal{B}$ is the least discriminating—it cannot distinguish any two systems from each other (proposition 2). By proposition 3 the perfect vision environment $\mathcal{V}$ is the most discriminating one.

The notion of discrimination will soon prove fundamental for our developments. We shall use it to design composition operators for behavioral properties, facilitating hierarchical modeling of product lines. Unfortunately the definition of the discrimination is rather abstract. The quantification over all systems, makes it infeasible to reason about it mechanically. To remedy this obstacle we introduce a new preorder on environments: a simulation for color-blind IOATSs.

Definition 7 *Let $\mathcal{E} = (Out, In, Gen_{\mathcal{E}}, Obs_{\mathcal{E}}, \xrightarrow{!}_{\mathcal{E}}, \xrightarrow{?}_{\mathcal{E}}, E^0)$ and $\mathcal{F} = (Out, In, Gen_{\mathcal{F}}, Obs_{\mathcal{F}}, \xrightarrow{!}_{\mathcal{F}}, \xrightarrow{?}_{\mathcal{F}}, F^0)$ be color-blind environments. A pair of binary relations, $R_1 \subseteq Gen_{\mathcal{E}} \times Gen_{\mathcal{F}}$ and $R_2 \subseteq Obs_{\mathcal{F}} \times Obs_{\mathcal{E}}$, constitutes a simulation between states of color-blind IOATSs iff $(E, F) \in R_1$ implies that*

$$\textit{whenever } E \xrightarrow{i!} e \textit{ then also } F \xrightarrow{i!} f \textit{ and } (f, e) \in R_2 \ ,$$

and $(f, e) \in R_2$ implies that whenever $f \xrightarrow{O_f?} F$

$$\textit{then also } e \xrightarrow{O_e?} E \textit{ and } O_f \subseteq O_e \textit{ and } (E, F) \in R_1 \ .$$

Let (R_1, R_2) be the largest such pair of relations (ordered by point-wise inclusion). A generator F simulates a generator E, written $E \leqslant F$, iff $(E, F) \in R_1$. An observer e simulates an observer f, written $f \leqslant e$, iff $(f, e) \in R_2$. An environment $\mathcal{F}$ simulates $\mathcal{E}$, written $\mathcal{E} \leqslant \mathcal{F}$, iff $E^0 \leqslant F^0$.

Let $\mathcal{E}$, $\mathcal{F}$ be color-blind environments as in the above definition. Let $\mathbb{S}$ be an endofunction on pairs of binary relations on states of these environments such

that:

$$\begin{aligned}\mathbb{S}(R_1, R_2) = (\{(E, F) \mid \forall i, e.\, \exists f. E \xrightarrow{i!} e \implies F \xrightarrow{i!} f \text{ and } (f, e) \in R_2\},\\ \{(f, e) \mid \forall O_f, F'.\, \exists O_e, E'.\, f \xrightarrow{O_f?} F' \implies e \xrightarrow{O_e?} E'\\ \text{and } O_f \subseteq O_e \text{ and } (E', F') \in R_1\})\end{aligned}$$

Proposition 4 *A pair of binary relations (R_1, R_2) constitutes a simulation between color-blind IOATSs iff $(R_1, R_2) \subseteq \mathbb{S}(R_1, R_2)$ (pointwise inclusion).*

Proof 2 *$\mathbb{S}$ is a monotonic endofunction on a complete lattice of pairs of binary relations. By Tarski's theorem $\mathbb{S}$ has the greatest fixpoint $\bigcap_{j=0}^{\infty} \mathbb{S}^j(Gen_{\mathcal{E}} \times Gen_{\mathcal{F}}, Obs_{\mathcal{F}} \times Obs_{\mathcal{E}})$, equal to the simulation of Def. 7. Since the fixpoint contains all simulations, it enables use of the known proof technique: to show that an IOATS simulates another, find any simulation relating them.* □

The simulation preorder can be established mechanically for finite state systems using state exploration [EMCGP99]. Thanks to the following central result, these techniques can also be used to verify discrimination properties:

Theorem 5 *For any two color-blind environments $\mathcal{E}$ and $\mathcal{F}$: $\mathcal{E} \sqsubseteq \mathcal{F}$ iff $\mathcal{E} \leqslant \mathcal{F}$.*

We prove the theorem at the state level, which generalizes directly to the IOATS level.

Definition 8 *Let $\mathcal{E}$ and $\mathcal{F}$ be color-blind environments with identical signatures and let $E \in Gen_{\mathcal{E}}$, $F \in Gen_{\mathcal{F}}$ be generators. The generator F is more discriminating than E, written $E \sqsubseteq F$, iff for all observers s_1, s_2 of all systems $\mathcal{S}_1$, $\mathcal{S}_2$ (compatible with $\mathcal{E}$, $\mathcal{F}$) $F \models s_1 \leqslant s_2$ implies $E \models s_1 \leqslant s_2$.*

Lemma 1 *For any generators $E \in Gen_{\mathcal{E}}$ and $F \in Gen_{\mathcal{F}}$ it holds that $E \leqslant F \Rightarrow E \sqsubseteq F$.*

Proof 3 *Let $\mathcal{S}_1$, $\mathcal{S}_2$ be systems compatible with $\mathcal{E}$, $\mathcal{F}$. Also let $E \leqslant F$, like in the lemma. We show that for any observers s_1, s_2 of $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively, $F \models s_1 \leqslant s_2$ implies $E \models s_1 \leqslant s_2$, or in other words that $(\leqslant_F) \subseteq (\leqslant_E)$. We proceed in two steps: first we introduce a $Gen_{\mathcal{E}}$-indexed family of relations R such that $(\leqslant_F) \subseteq R_E$, second we argue that R_E is an $\mathcal{E}$-relativized simulation, so $R_E \subseteq (\leqslant_E)$.*

$$R_E = \{(s_1, s_2) \in Obs_{\mathcal{S}_1} \times Obs_{\mathcal{S}_2} \mid \exists F' \in Gen_{\mathcal{F}}.\, E \leqslant F' \wedge F' \models s_1 \leqslant s_2\}$$

First step: $(\leqslant_F) \subseteq R_E$, since $E \leqslant F$. Second step: take s_1, s_2 and E' such that $(s_1, s_2) \in R_{E'}$. Let $E' \xrightarrow{i!} e'$ and $e' \xrightarrow{O_e?} E''$ and $s_1 \xrightarrow{i?} S_1$ and $S_1 \xrightarrow{o_1!} s_1'$ and

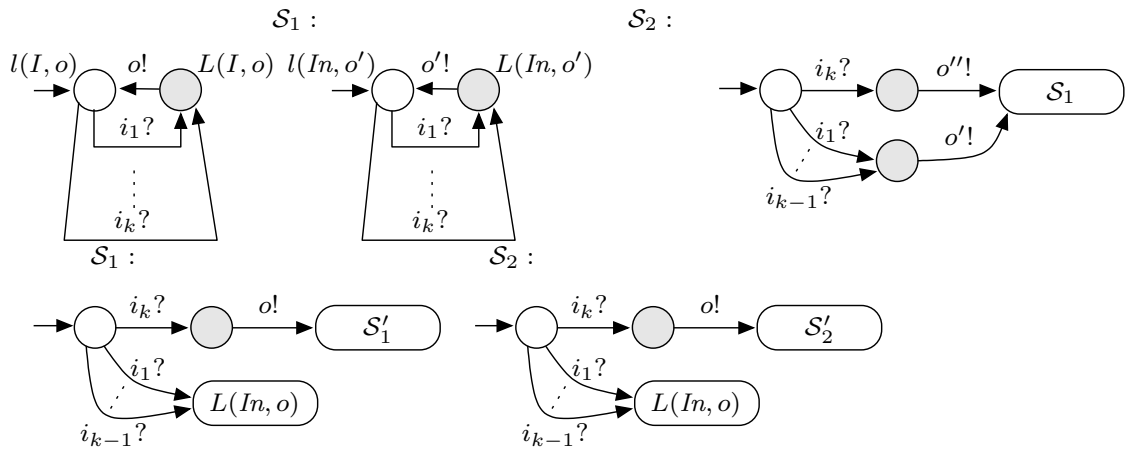


Figure A.13: a) $\mathcal{L}(I,o)$. b) Counter example systems $\mathcal{S}_1$ and $\mathcal{S}_2$. c) $\mathcal{S}_1$ and $\mathcal{S}_2$ created in the inductive step of the proof of lemma 2. Although all these examples assume that $In = \{i_1, \ldots, i_j\}$, the finiteness of In is only a visualization convention and is not relied upon in the proofs.

$o_1 \in O_e$. We need to find S_2, o_2, s_2' such that $s_2 \xrightarrow{i?} S_2$ and $S_2 \xrightarrow{o_2!} s_2'$ and $o_2 \in O_e$ and $(s_1', s_2') \in R_{E''}$. But since $(s_1, s_2) \in R_{E'}$ there must exists F' such that $F' \models s_1 \leqslant s_2$ and $E' \leqslant F'$. The latter means that there exist f', O_f, F'' such that $F' \xrightarrow{i!} f'$ and $f' \xrightarrow{O_f?} F''$ and $o_1 \in O_f \subseteq O_e$ and $E'' \leqslant F''$, which combined with the former implies that $s_2 \xrightarrow{i?} S_2$ and $S_2 \xrightarrow{o_2!} s_2'$ and $o_2 \in O_f \subseteq O_e$. It remains to be shown that $(s_1', s_2') \in R_{E''}$, which follows from the definition of R, as $E'' \leqslant F''$.

Lemma 2 *For any generators $E \in Gen_{\mathcal{E}}$ and $F \in Gen_{\mathcal{F}}$ it holds that $E \sqsubseteq F \Rightarrow E \leqslant F$.*

For a set of inputs $I \subseteq In$ and an output $o \in Out$ define a looping system $\mathcal{L}(I,o)$ (Fig. A.13). Let $L(I,o)$ denote a generator and $l(I,o)$ denote an observer. In $\mathcal{L}(I,o)$ there is an observation transition from $l(I,o)$ to $L(I,o)$ for every $i \in I$ and a single generation transition $L(I,o) \xrightarrow{o!} l(I,o)$.

Proof 4 *We prove the contrapositive: for all $\mathcal{E}$,$\mathcal{F}$ and E, F, their generators, $E \not\leqslant F$ implies $E \not\sqsubseteq F$ (there exist systems $\mathcal{S}_1, \mathcal{S}_2$ and their observers s_1, s_2 such that $F \models s_1 \leqslant s_2$ but $E \not\models s_1 \leqslant s_2$).*

Since $E \not\leqslant F$ then there exists $n \geq 1$ such that $(E,F) \in \bigcap_{j=0}^{n-1} \mathbb{S}^j(Gen_{\mathcal{E}} \times Gen_{\mathcal{F}}, Obs_{\mathcal{F}} \times Obs_{\mathcal{E}})$ and $(E,F) \notin \bigcap_{j=0}^{n} \mathbb{S}^j(Gen_{\mathcal{E}} \times Gen_{\mathcal{F}}, Obs_{\mathcal{F}} \times Obs_{\mathcal{E}})$.
1°. if $n = 1$ then there exist input i_k, observation class O_f and observers e,f such that $E \xrightarrow{i_k!} e$ and $F \xrightarrow{i_k!} f$ and $f \xrightarrow{O_f!} F'$, but for all transitions outgoing from e, $e \xrightarrow{O_e?} E'$, we have that $O_f \not\subseteq O_e$. Because of this and the fact that the observation classes of e form a partitioning of Out (see (A.2)), there exist two distinct observation classes O_e', O_e'' of e, such that $O_e' \cap O_f \neq \emptyset$ and $O_e'' \cap O_f \neq \emptyset$. Let o' be an arbitrary element from $O_e' \cap O_f$ and similarly $o'' \in O_e'' \cap O_f$. We shall now

construct our two systems $\mathcal{S}_1$ and $\mathcal{S}_2$. The idea is that the first steps of $\mathcal{S}_1$ and $\mathcal{S}_2$ differ insufficiently to be distinguished by f, but sufficiently for e to distinguish them. In the subsequent steps both systems behave identically. Let $\mathcal{S}_1$ just consist of $l(In, o')$ and $\mathcal{S}_2$ be as on Fig. A.13b. It is easy to observe that $F \models s_1 \leqslant s_2$, but $E \not\models s_1 \leqslant s_2$.
2°. Inductive step. Now consider that $n > 1$. In a similar manner as above we would like to construct two systems which violate E-relativized simulation in the nth step on the very trace, on which E and F disagree. On all other traces of length n, and all longer traces they should behave identically.

Consider the prefixes of the execution witnessing $E \not\leqslant F$: $E \xrightarrow{i_k!} e, e \xrightarrow{O_e?} E'$ and $F \xrightarrow{i_k!} f$, $f \xrightarrow{O_f?} F'$. Since $n > 1$, $O_f \subseteq O_e$ and there exists a violation of simulation between E' and F' in $n-1$ steps. By our induction hypothesis there exist systems $\mathcal{S}'_1$ and $\mathcal{S}'_2$ and states thereof S'_1 and S'_2 such that $F \models S'_1 \leqslant S'_2$ and $E \not\models S'_1 \leqslant S'_2$. We create a new pair of systems $\mathcal{S}_1$ and $\mathcal{S}_2$ by adding new initial observers s_1, s_2 and generators S_1, S_2 with transitions $s_1 \xrightarrow{i_k?} S_1$, $S_1 \xrightarrow{o!} s'_1$ and $s_2 \xrightarrow{i_k?} S_2$, $S_2 \xrightarrow{o!} s'_2$, where $o \in O_f \subseteq O_e$ and s'_1 and s'_2 are the initial states of $\mathcal{S}'_1$ and $\mathcal{S}'_1$. Both for s_1 and s_2 we also add transitions for all inputs different than i_k to $l(In, o)$. See Fig. A.13c. It is not hard to see that $F \models s_1 \leqslant s_2$, but $E \not\models s_1 \leqslant s_2$. □

5 Composition of Behavioral Properties

Typical code generators do not use any context information, assuming that the model is combined with the perfect vision environment $\mathcal{V}$. Another extreme would be a program synthesis tool requiring a precise environment model, imposing a significant burden on engineers. We propose light-weight, composable, partial specifications of environments in the form of behavioral properties like: that certain events always come interleaved (e.g. on/off switch), or that there is causality between an input and an output (e.g. a timer only timeouts after it has been started). Each property can be expressed as a simple color-blind IOATS. In this section we consider ways of composing such properties in a conjunctive and disjunctive manner. Conjunctions express adding up inabilities of environments. If one event is unobservable and another event is unobservable then both are unobservable. Disjunctions express adding up abilities. If one event is observable or another is observable, then both might be observable. This means that conjunction decreases discriminative power, making observation classes coarser in the transition systems representing properties, while disjunction increases discriminative power, making observation classes finer. It turns out that the suitable semantics for both connectives can be constructed using greatest lower bounds (glbs) and least upper bounds (lubs) with respect to the discrimination preorder $\sqsubseteq$.

As said before, every observer e of a color-blind IOATS induces a partitioning

of *Out* into observation classes. Let us denote this partitioning by P_e. The set of all equivalence relations over *Out* ordered by inclusion forms a complete lattice (and hence the set of all partitionings). Consequently for any set of partitionings $\{P_k\}_{k\in L}$ there exist the greatest lower bound $\bigsqcap_{k\in L} P_k$, which is the coarsest partitioning finer than any of P_k and the least upper bound $\bigsqcup_{k\in L} P_k$, which is the finest partitioning coarser than all P_k.

The composition is defined for environments with the same I/O signatures. We consider two kinds of composition: a sum and a product. Due to the alternating nature of communication in our setup, sums and products alternate too: a sum of generators evolves into a product of observers, and dually a sum of observers evolves into a product of generators.

Definition 9 *Let* $\mathcal{E} = (Out, In, Gen_{\mathcal{E}}, Obs_{\mathcal{E}}, \xrightarrow{!}_{\mathcal{E}}, \xrightarrow{?}_{\mathcal{E}}, E^0)$ *and* $\mathcal{F} = (Out, In, Gen_{\mathcal{F}}, Obs_{\mathcal{F}}, \xrightarrow{!}_{\mathcal{F}}, \xrightarrow{?}_{\mathcal{F}}, F^0)$ *be color-blind environments. We define their sum to be a color-blind IOATS* $\mathcal{E}+\mathcal{F} = (Out, In, Gen_{\mathcal{EF}}, Obs_{\mathcal{EF}}, \xrightarrow{!}, \xrightarrow{?}, \{E^0, F^0\})$*, where* $Gen_{\mathcal{EF}} = \mathcal{P}(Gen_{\mathcal{E}} \cup Gen_{\mathcal{F}})$*,* $Obs_{\mathcal{EF}} = \mathcal{P}(Obs_{\mathcal{E}} \cup Obs_{\mathcal{F}})$ *and* $\xrightarrow{!}$*,* $\xrightarrow{?}$ *are defined by recursively applying the SG and PO rules given shortly.*

Sums intuitively correspond to disjunctions of properties. The composition is synchronous: all composed generators take identical steps simultaneously. From the system's perspective a single input is generated. A sum should be as discriminating as any of the summands. For this reason after taking a generation transition over an input i, the sum advances to a product of generators reachable by i from any of the summands. This product, which also embeds determinization, builds the coarsest observation relation that is *finer* than any of the original observation relations, guaranteeing that indeed the constructed observer is as discriminative as necessary:

$$\frac{E_1 \xrightarrow{i!} e_1 \ \ldots \ E_n \xrightarrow{i!} e_n}{\sum_{k=1}^{n} E_k \xrightarrow{i!} \prod_{k=1}^{n} e_k}\ \text{SG}$$

$$\frac{O \in \prod_{k=1}^{n} P_{e_k} \quad \mathbb{E} = \{E \mid \exists 1 \le k \le n. \exists O' \subseteq Out. e_k \xrightarrow{O'?} E \wedge O \subseteq O'\}}{\prod_{k=1}^{n} e_k \xrightarrow{O?} \sum \mathbb{E}}\ \text{PO}$$

Observation classes in the product of observers (PO) are finer than observation classes of any of the composed processes. Whenever any output o is observed by the result of the composition we advance to the state $\mathbb{E}$ composed of states reachable by o from all e_k's. Since O is finer than some class in any of these observers there is always exactly n such reachable generators.

The product of two environment IOATSs is defined as follows.

Definition 10 *Let* $\mathcal{E} = (Out, In, Gen_{\mathcal{E}}, Obs_{\mathcal{E}}, \xrightarrow{!}_{\mathcal{E}}, \xrightarrow{?}_{\mathcal{E}}, E^0)$ *and* $\mathcal{F} = (Out, In, Gen_{\mathcal{F}}, Obs_{\mathcal{F}}, \xrightarrow{!}_{\mathcal{F}}, \xrightarrow{?}_{\mathcal{F}}, F^0)$ *be color-blind environments. Their product is an IOATS* $\mathcal{E} \times \mathcal{F} = (Out, In, Gen_{\mathcal{EF}}, Obs_{\mathcal{EF}}, \xrightarrow{!}, \xrightarrow{?}, \{E^0, F^0\})$*, where* $Gen_{\mathcal{EF}} = \mathcal{P}(Gen_{\mathcal{E}} \cup Gen_{\mathcal{F}})$*,* $Obs_{\mathcal{EF}} = \mathcal{P}(Obs_{\mathcal{E}} \cup Obs_{\mathcal{F}})$ *and transition relations* $\xrightarrow{!}$*,* $\xrightarrow{?}$ *are defined by recursively applying the PG and SO rules given shortly.*

A product of generators corresponds to a conjunction of properties (or synchronous composition in CSP [Hoa85]). Again the generator rule is very simple and synchronous, but the observer rule this time builds the finest observation relation that is *coarser* than any of the factors:

$$\frac{E_1 \xrightarrow{i!} e_1 \ \ldots \ E_n \xrightarrow{i!} e_n}{\prod_{k=1}^{n} E_k \xrightarrow{i!} \sum_{k=1}^{n} e_k} \ \text{PG}$$

$$\frac{O \in \bigsqcup_{k=1}^{n} P_{e_k} \quad \mathbb{E} = \{E | \exists 1 \leq k \leq n. \exists O' \subseteq O. e_k \xrightarrow{O'?} E\}}{\sum_{k=1}^{n} e_k \xrightarrow{O?} \prod \mathbb{E}} \ \text{SO}$$

Observation classes in the sum of observers (SO) are coarser than classes of any of the composed observers. The transition relation follows to those generators that can be reached by any output belonging to such an extended class. The size of $\mathbb{E}$ can exceed the number of original observers n.

The result of a composition is a well-formed color-blind IOATS enjoying the following essential property:

Theorem 6 $\sum_{k=1}^{n}\{\mathcal{E}_k\}$ *is the least environment with respect to* $\leqslant$*, which simulates all summands, while* $\prod_{k=1}^{n}\{\mathcal{E}_k\}$ *is the greatest environment with respect to* $\leqslant$*, which is simulated by all the factors.*

Since discrimination and simulation coincide (Thm. 1) $\sqsubseteq$ can replace $\leqslant$ in the above theorem: *The sum of environments is the least discriminating environment, more discriminating than each of the summands. The product is the most discriminating environment, less discriminating than each of the factors.* These in turn are standard expectations about conjunction and disjunction. A conjunction (product) of two properties expressing inability to observe two behaviors, will result in a property expressing inability to observe either. Disjunction (sum) of two properties expressing ability to observe something, results in a property expressing the ability to observe both. See example on Fig. A.14.

Proof 5 *We shall show the theorem on the state level (the result on the IOATSs level follows directly). First show that* $\forall k = 1 \ldots n.\ E_k \leqslant \sum_{k=1}^{n} E_k$*. This is in fact the case because the product of observers creates observation classes which are always subsets of original classes (partitioning of the original classes). More*

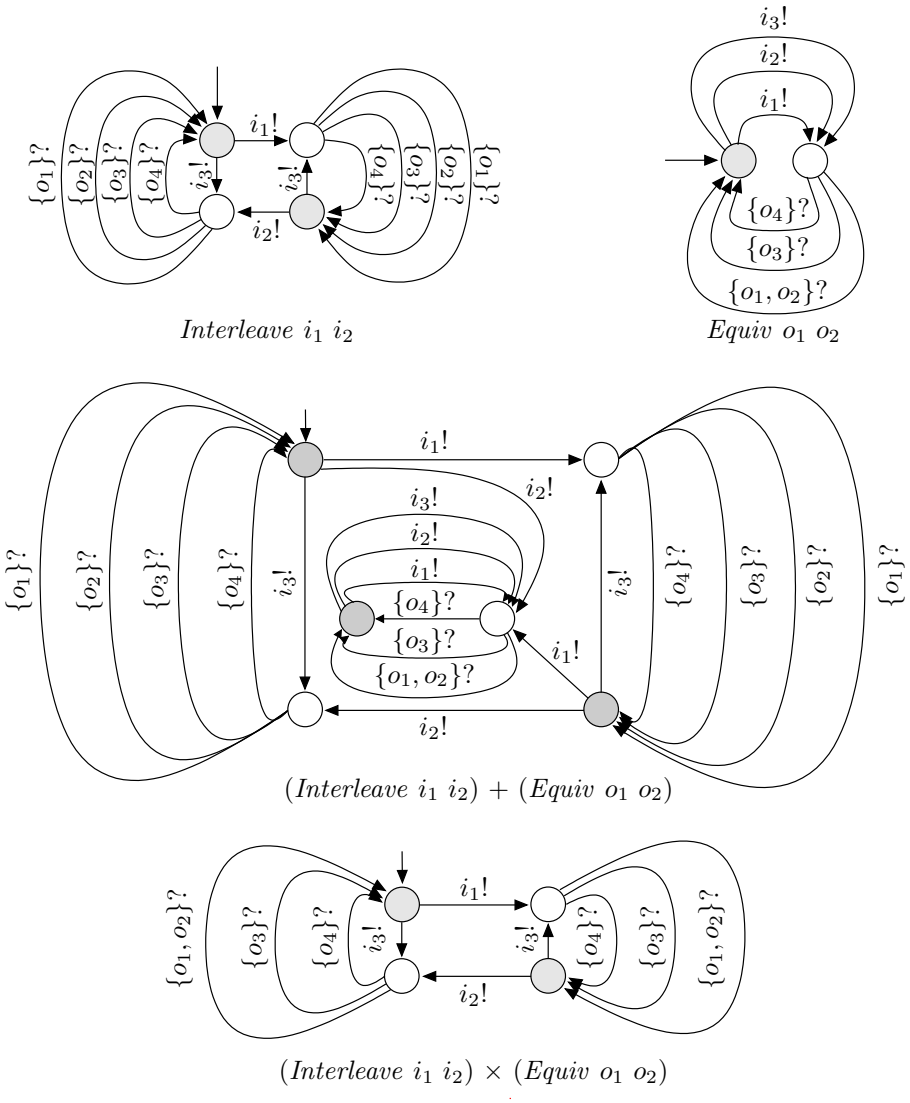


Figure A.14: Environments *Interleave* i_1 i_2 (left) and *Equiv* o_1 o_2 (right), their sum (top) and product (bottom). Transitions to the blind observer **b** are supressed. The product can only generate what both of the factors could generate and distinguish only what both of them could distinguish. The sum can generate what any of the summands could generate and observe what any of them could observe. In particular o_1 and o_2 are distinguished in the traces for which the *Interleave* property is preserved and not otherwise.

formally it can be argued by showing that the pair of relations (R_1, R_2)*, defined as below forms a simulation on environments:*

$$R_1 = \left\{ \left(E_j, \sum_{k \in I} E_k \right) \;\middle|\; \begin{array}{c} \textit{for any finite } I \textit{ and gen-} \\ \textit{erators } \{E_k\}_{k \in I} \textit{ and } j \in I \end{array} \right\}$$
$$R_2 = \left\{ \left(\prod_{k \in I} e_k, e_j \right) \;\middle|\; \begin{array}{c} \textit{for any finite } I \textit{ and obser-} \\ \textit{vers } \{e_k\}_{k \in I} \textit{ and } j \in I \end{array} \right\} \tag{A.4}$$

It remains to show that for all such generators F *that* $\forall k = 1 \ldots n.\ E_k \leqslant F$*, it holds that also* $\sum_{k=1}^{n} E_k \leqslant F$*. This in turn is achieved by showing that the pair of relations* (R_3, R_4) *forms a simulation on environments, where:*

$$R_3 = \left\{ \left(\sum_{k \in I} E_k, F \right) \;\middle|\; \begin{array}{c} \forall \textit{ finite } I.\ \forall \{E_k\}_{k \in I}. \\ \forall F.\ \forall k \in I. E_k \leqslant F \end{array} \right\}$$
$$R_4 = \left\{ \left(f, \prod_{k \in I} e_k \right) \;\middle|\; \begin{array}{c} \forall \textit{ finite } I.\ \forall \{e_k\}_{k \in I}. \\ \forall f. \forall k \in I.\ f \leqslant e_k \end{array} \right\} \tag{A.5}$$

The proof of the case for products of generators is dual.□

6 Toward Realistic Design Languages

Until now we have assumed that outputs of systems are atomic. This assumption however often does not hold for realistic languages, which typically support structured output: sets, multisets, sequences or even sequences of sets of atomic actions produced in a single step. We will study two groups of languages. We have successfully applied our framework to the semantics of languages producing sets (state/event systems of section 2, Harel's statecharts [Har87], synchronous languages [Ber00]) and sequences (Java Card [Sun], UML state diagrams [Obj99]). Each of these language groups, set based and sequence based, will be discussed in the two following subsections. The set based version is further demonstrated by example in Section 7.

From now on assume a finite set of environment events *Event* and a finite set of atomic output actions *Action*. In order to be able to handle realistic languages we need to instantiate our framework for a given reaction style. This includes not only giving mappings from *Event* and *Action* to *In* and *Out*, but also proposing suitable representation for observation classes and computing bounds on classes.

6.1 Set based

State/event models of section 2 are synchronous and non-blocking, so they meet all the assumptions of our framework. The set of abstract events $In = Event$,

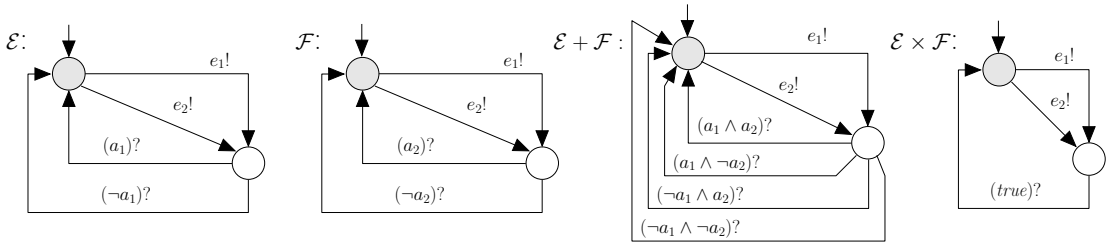


Figure A.15: Left: environments $\mathcal{E}$ and $\mathcal{F}$, suitable for executing set-based reactive systems. Right: the sum and product of $\mathcal{E}$ and $\mathcal{F}$. Observational classes computed according to propositions 7 (for sum) and 8 (for product)

while the set of abstract outputs contains all possible subsets of *Action*.

$$In = Event \qquad Out = \mathcal{P}(Action)$$

Each state configuration $s \in State$ corresponds to a single observer at the abstract level, while a new generator is added for each pair of configurations (in practice it suffices to consider pairs of configurations related in the global transition relation):

$$Obs = State \qquad Gen = State \times State$$

Finally for every global transition $s \xrightarrow{e \quad o} s'$ at the model level we introduce a single observation transition $s \xrightarrow{e?} (s, s')$ and a single generation transition $(s, s') \xrightarrow{o!} s'$ in the abstract IOATS. Remaining states are not related.

These definitions allow us to use the framework of previous sections and model environments for State Event Systems as color-blind IOATSs. Note though, that, since *Out* is a powerset now, observation classes are not simply sets of outputs, but sets of subsets of *Action*. How should these classes be specified and represented? How can we compute the greatest lower bounds (glbs) and least upper bounds (lubs) on partitionings in this domain to efficiently obtain sums and products of IOATSs?

Subsets of finite sets, such as *Action*, are conveniently described with propositional formulæ. For each formula φ over variables representing elements of *Action* consider a corresponding set of its satisfiable assignments. Each assignment describes a set of actions, or a single output. We can use propositional formulæ instead of explicit enumerations as specifications of observational classes. More importantly we can efficiently implement them symbolically using Reduced Ordered Binary Decision Diagrams, or BDDs [Bry86].

We still need to make sure that classes represented on transitions leaving from a single observer are indeed disjoint and form a partitioning (see (A.2)). To achieve this we require that all corresponding formulæ are mutually exclusive and that they add up to the complete universum. For a set of formulæ $\varphi_1, \ldots, \varphi_n$ labeling all distinct transitions outgoing from a single observer state the following

two conditions must hold:

$$\forall i, j \in \{1..n\}.\ i \neq j \ \Rightarrow\ \varphi_i \wedge \varphi_j \equiv \mathit{false} \tag{A.6}$$

$$\varphi_1 \vee \ldots \vee \varphi_n \equiv \mathit{true} \tag{A.7}$$

These conditions are feasible to verify computationally, especially easily using a BDD engine, or a SAT-solver.

Syntactically correct environments can be combined in sums and products using the operational rules presented in section 5. In particular the rules for observers rely on the existence of glbs and lubs for partitionings. Even though these glbs and lubs exists, we still need to give an efficient way to compute them:

Proposition 7 *Consider two equivalence relations $\sim_\varphi$ and $\sim_\psi$ defined on $\mathcal{P}(Action)$, such that the observation classes of $\sim_\varphi$ are described by formulæ $\varphi_1, \ldots, \varphi_m$ and observation classes of $\sim_\psi$ are described by formulæ $\psi_1, \ldots, \psi_n$. Then the equivalence relation $\sim_\varphi \sqcap \sim_\psi$ is characterized by formulæ:*

$$\{\varphi_i \wedge \psi_j | i = 1 \ldots m,\ j = 1 \ldots n\}\ .$$

Obviously some of the new observational classes may be empty, since usually not all conjunctions are satisfiable. Unsatisfiable formulæ can be eliminated since corresponding BDDs automatically reduce to *false*.

The lub of two equivalence relations is a transitive closure of the union of these two relations. The computation of this transitive closure is realized by the classic Union-Find algorithm (see [CLRS01, chapter 21]) applied to the observation classes of both relations. Any two overlapping classes should be merged until no more classes overlap. An overlapping occurs if the conjunction of the two respective formulæ is satisfiable. A union of the class represented by φ with a class represented by ψ corresponds to replacement of the two formulæ with a disjunction $\varphi \vee \psi$ of the two.

Proposition 8 *Let $\sim_\varphi$, $\sim_\psi$ be equivalence relations on $\mathcal{P}(Action)$, such that their observation classes are described by formulæ $\varphi_1, \ldots, \varphi_m$ and $\psi_1, \ldots, \psi_n$ respectively. Then the equivalence relation $\sim_\varphi \sqcup \sim_\psi$ is characterized by formulæ computed using the* Union-Find *algorithm applied to the set $\{\varphi_1, \ldots, \varphi_m, \psi_1, \ldots, \psi_n\}$, where two formulæ are unifiable, if their conjunction is satisfiable, and disjunction is the union operation.*

Fig. A.15 presents examples of environments with observational classes represented by propositional formulæ together with their sum and product computed using the intersection and the Union-Find algorithm.

We remark, that a nearly identical adaptation allows applications of our framework to other set-based languages including many hardware description languages, synchronous languages [Ber00] and Harel's statecharts [Har87].

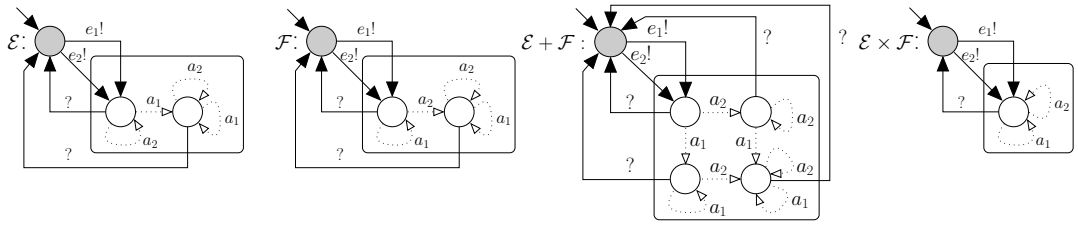


Figure A.16: Environments $\mathcal{E}$ and $\mathcal{F}$ observing sequences, their sum and product.

6.2 Sequence based

Let us now turn from systems producing outputs structured as sets towards systems that produce outputs structured as sequences of atomic actions—for example UML state diagrams. Now each observation transition of the system awaits a single input from the *Event* set, while each generation transition produces an output which is a finite sequence of actions from *Action*:

$$In = Event \text{ and } Out = Action^* \ .$$

The first step in adapting the theory is linking the concrete states of models (for example state configurations in statecharts, or variable store in Java Card) to abstract states of the IOATS. This can normally be done in a direct way (at least for finite state models) in the same spirit as in the previous section. Subsequently the observation and generation relations must be extracted from the semantics of the language in question. Observation classes on the environment side (color-blind) become sets of sequences of actions. Partitioning of $Action^*$ into classes that are regular languages can be described by a finite automaton.

Definition 11 *A classifier DFA over alphabet A is a quadruple $c = (S, A, s, \longrightarrow)$, where S is a finite set of states, A is a finite set of symbols, $s \in S$ is an initial state and $\longrightarrow \in S \to A \to S$ is an input-enabled transition function, meaning that for every $s \in S$ function $\longrightarrow(s)$ is defined for each element of its domain A. We usually write $s \xrightarrow{a} s'$ instead of $\longrightarrow(s)(a) = s'$.*

A classifier DFA consecutively applies $\longrightarrow$ to a state and the head of the input sequence obtaining a new state and input sequence. An execution over a list of symbols $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ \ldots \ \xrightarrow{a_n} s_n$ is abbreviated with $s \xrightarrow{a_1 \ldots a_n}{}^* s_n$.

Definition 12 *Let $c = (S, A, s, \longrightarrow)$ be a classifier. Sequences $\sigma_1, \sigma_2 \in A^*$ are equivalent with respect to c if both advance c to the same state: $\exists s'.s \xrightarrow{\sigma_1}{}^* s' \wedge s \xrightarrow{\sigma_2}{}^* s'$.*

The equivalence with respect to a classifier is an equivalence relation and partitions A^* into a finite set of classes, isomorphic with the reachable states.

For a classifier $e = (S_e, Action, s_e, \rightarrow_e)$ consider a mapping of its states to generators $\gamma_e : S_e \rightarrow Gen$. Each observer of the environment comprises a classifier and a generator mapping. Environments advance from an observer (e, γ_e) to a generator $\gamma_e(s)$ if it observes a sequence σ advancing the classifier to a state s:

$$(e, \gamma_e) \xrightarrow{\{\sigma \mid s_e \xrightarrow{\sigma}{}^* s\}\ ?} \gamma_e(s) \ .$$

Fig. A.16 shows two color-blind IOATSs$\mathcal{E}$ and $\mathcal{F}$ of signature: $Event = \{e_1, e_2\}$ and $Action = \{a_1, a_2\}$. $\mathcal{E}$ distinguishes reactions containing at least one occurrence of a_1 from those not containing a_1 at all. Similarly $\mathcal{F}$ distinguishes between sequences containing at least one a_2 from those not containing a_2 at all. Observers are drawn as boxes containing classifier DFAs. Classifier transitions are represented as dotted arrows to distinguish them from IOATS transitions.

The product of classifiers is a central construction in computing products of observers, supporting composition of environments:

Definition 13 *Let $e = (S_e, A, s_e, \rightarrow_e)$ and $f = (S_f, A, s_f, \rightarrow_f)$ be classifiers. A product of e and f is a classifier $e \otimes f = (S_e \times S_f, A, (s_e, s_f), \rightarrow)$, where $(s_e, s_f) \xrightarrow{a} (s'_e, s'_f)$ if $s_e \xrightarrow{a} s'_e$ and $s_f \xrightarrow{a} s'_f$.*

Proposition 9 *Let $\sim_e$ and $\sim_f$ be two equivalences on $Action^*$ induced by classifiers e and f. Their greatest lower bound $\sim_e \sqcap \sim_f$ exists and is induced by $e \otimes f$.*

Fig. A.16 presents the sum $\mathcal{E} + \mathcal{F}$ obtained by application of operational rules of section 5 (SG,PO) and the above proposition. $\mathcal{E} + \mathcal{F}$ distinguishes four classes of outputs: an empty sequence, sequences consisting of occurrences of a_1, consisting of occurrences of a_2, and containing occurrences of both a_1 and a_2.

As we have seen before, the least upper bound of two partitionings $\sim_e \sqcup \sim_f$ is usually computed using a UNION-FIND algorithm, which unifies any two overlapping classes, until all classes are disjoint. In this case classes are represented by states in the classifiers e and f. We need to apply the algorithm to states of e and f, ultimately producing a classifier, whose states are sets of states of f and e. The two classes s_1 and s_2 overlap, whenever there is an output sequence, that can advance one classifier to a state in s_1, and the other classifier to a state in s_2. The initial set of classes is given by reachable states of the product classifier $e \otimes f$:

i. $S := \{\{e_i, f_j\} \mid (e_i, f_j)$ is reachable in $e \otimes f\}$.

ii. If there exist $s_1, s_2 \in S$ such that $s_1 \cap s_2 \neq \emptyset$ then $S := S \setminus \{s_1, s_2\} \cup \{s_1 \cup s_2\}$.

iii. Repeat (ii) until no more classes can be unified.

The final value of S is the set of states of the new classifier DFA. The initial state is the class that contains initial states of e and f (note that both of them will be in the same class). The transition function $\rightarrow$ is a sum of transition functions $\rightarrow_e$ and $\rightarrow_f$ lifted to sets of states. For $s_1, s_2 \in S$:

$$s_1 \xrightarrow{a} s_2 \text{ if } \exists . p_1 \in S_1 . \exists p_2 \in S_2 .\ p_1 \xrightarrow{a}_e p_2 \text{ or } p_1 \xrightarrow{a}_f p_2$$

The following proposition claims that this function is well-defined, deterministic and input-enabled:

Proposition 10 *Let $s_1, s_2 \in S$ be any two of the sets of states (not necessarily distinct) constructed with the above algorithm. Then for any states $p_1, p_2 \in s_1$, $p_1', p_2' \in s_2$ of the original classifiers and any symbol a: $p_1 \xrightarrow{a}_1 p_1'$ and $p_1' \in s_2$ iff $p_2 \xrightarrow{a}_2 p_2'$ and $s_2' \in s_2$, where $\longrightarrow_i$ denotes $\longrightarrow_e$ if $s_i \in S_e$ or $\longrightarrow_f$ if $s_i \in S_f$.*

It follows that the classifier $g = (S, A, s, \longrightarrow)$ constructed above is a well defined classifier DFA. Moreover, the observation classes that it induces are coarser than any class of $\sim_e$ and $\sim_f$. Due to the properties of the UNION-FIND algorithm, $\sim_g$ is actually the least equivalence encompassing both $\sim_e$ and $\sim_f$:

Proposition 11 *Let $\sim_e$ and $\sim_f$ be equivalences over $Action^*$, induced by classifiers $e = (S_e, Action, s_e, \longrightarrow_e)$ and $f = (S_f, Action, s_f, \longrightarrow_f)$. The equivalence $\sim_e \sqcup \sim_f$ is induced by a classifier g such that its states are computed applying the* UNION-FIND *algorithm to the set*

$$\{\ \{e_i, f_j\} \mid (e_i, f_j) \textit{ reachable in } e \otimes f\ \}\ ,$$

where two sets s_1,s_2 are unifiable if $s_1 \cap s_2$ is not empty. The union operation is a set union, the initial state is the set containing initial states of e and f, and the transition function is a sum of transition functions lifted to sets of states.

The rightmost IOATS on Fig. A.16 is a product of $\mathcal{E}$ and $\mathcal{F}$ obtained by application of the composition rules from section 5 (PG,SO) and the above algorithm. This product gives rise to the observer which does not distinguish any sequences.

7 Environment Driven Specialization

We shall now broaden the meaning of a model of a system to encompass a family of systems, and let it represent functionality, which in its entire richness may not be present in any of the actual members being produced. Particular family members will be specified using models of environments, and derived by transformations preserving relativized equivalence in a given color-blind environment. Each transformation can only be applied if its application precondition is satisfied. We face two proof obligations here. The first is a manual proof of correctness

of the transformation itself, which can be done ahead of time. The second is the application precondition satisfaction check, which takes place at the specialization time. This proof should be obtained automatically using one of the available technologies (type checking, static analysis and model-checking). In this section we firstly formulate correctness condition for transformations and then informally demonstrate a product line derivation scenario, hinting at what techniques could be used to make such automatic derivation viable.

Let T be a model transformation, $\mathcal{M}$ a family model and $\mathcal{E}$ an environment defining a specific family member. Then we will say that T is correct iff the original model and the derived member are in $\mathcal{E}$-relativized two-way simulation relation. $T(\mathcal{M}, \mathcal{E}) \lesssim_{\mathcal{E}} \mathcal{M}$. This means that $\mathcal{E}$ cannot distinguish the behavior of the two systems.

We will now present a family of environment specifications and a corresponding family of alarm clocks derived from the original alarm clock model using the environments. The transition relation of state/event systems (see section 2) produces sets of actions during a single reaction step. In such a setting the observational classes of environments become sets of sets (powersets) of actions.

For a set $A \subseteq \mathit{Action}$ let *ignore* A denote observation classes, which ignore elements of A, but distinguish all the other actions:

$$\mathit{ignore}\ A = \left\{\ \{o \cup o' | o' \in \mathcal{P}(A)\}\ \middle|\ o \in \mathcal{P}(\mathit{Action} \setminus A)\right\}$$

Note that ignoring the empty set, *ignore* $\emptyset$, means observing all differences in outputs. Another abbreviation *equiv* A denotes observation classes, which are unable to distinguish between any actions in A:

$$\mathit{equiv}\ A = \left\{\ \{o \cup o' | o' \in \mathcal{P}(A) \setminus \emptyset\}\ \middle|\ o \in \mathcal{P}(\mathit{Action} \setminus A)\right\} \cup\ \mathcal{P}(\mathit{Action} \setminus A)$$

We shall begin with stating general requirements, which hold for all the environments used to execute the alarm clock. These general requirements usually reflect the physical nature of actuators and sensors. In the case of our alarm clock events *dark*/*bright* and *snooze*/*snoozeR* are always generated in an alternating fashion:

$$\mathcal{E}_0 = \mathit{Interleave\ snooze\ snoozeR}\ \wedge\ \mathit{Interleave\ dark\ bright}\ .$$

Fig. A.8 demonstrates how *Interleave* could be defined using a set-based semantics.

In section 2 the most restricted family member $\mathcal{C}_4$ was introduced in figure Fig. A.7. Here we will first introduce some family members that are less restricted.

The least restricted member of the family $\mathcal{C}_1$ is shown in Fig. A.4. This model operates in an environment, which becomes blind for the *lightOn* action right after generating the *snooze* event. Formally the environment in which $\mathcal{C}_1$ behaves identical to $\mathcal{C}_0$ is $\mathcal{E}_1 = \mathcal{E}_0 \wedge \mathcal{E}'$, where $\mathcal{E}'$ is defined on Fig. A.9.

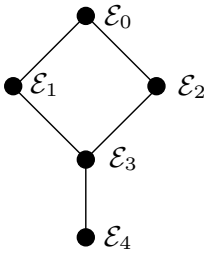


Figure A.17: Relationships between the environments.

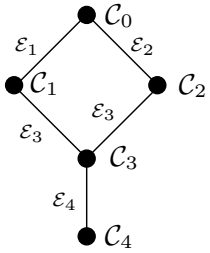


Figure A.18: Relationships between the product variants.

Fig. A.5 presents a new clock $\mathcal{C}_2$, which is devoid of the actual snooze function. The user of this clock can still press the snooze button, but the only effect it has is turning the backlight on for a short while. This user becomes blind to *beepOn* and $beepOff$ actions initiated by the *snooze* and *snoozeTO* events. Formally $\mathcal{E}_2 = \mathcal{E}_0 \wedge \mathcal{E}''$, where $\mathcal{E}''$ is defined on Fig. A.10.

The third clock variant $\mathcal{C}_3$ is a combination of $\mathcal{C}_1$ and $\mathcal{C}_2$. It has neither the snooze function nor the snooze activated backlight. We obtain it by specialization against the $\mathcal{E}_3$ environment, where $\mathcal{E}_3 = \mathcal{E}_1 \wedge \mathcal{E}_2$. The model is presented on Fig. A.6. Note that $\mathcal{C}_3$ still needs a snooze button, which exhibits a slight anomaly in turning on the glow mode, namely that the glow mode will not be activated, while this button is pressed. This is a perfectly correct reminiscence of our original model, which could be easily remedied by adding another constraint to the environment, that event *snooze* never occurs.

We would like to consider yet another restriction of the clock behavior. The clock denoted $\mathcal{C}_4$, shall be deprived of the glowing mode (Fig. A.7). The glow-mode lamp is not installed and the *glow* action is reimplemented to turn off the main lamp instead. A corresponding environment $\mathcal{E}'''$ is defined on Fig. A.11. This environment is itself interesting as it specifies a less shiny alarm clock, which may find its happy customers. Nevertheless, we decided to combine its characteristics with the restrictions of $\mathcal{E}_3$, giving rise to an even more simple alarm clock with neither the snooze related functions nor the glow mode: $\mathcal{E}_4 = \mathcal{E}_3 \wedge \mathcal{E}'''$.

One can describe surprisingly many more reasonable variants even for such a simple system. Figures A.17–A.18 present an overview of environments and systems in our product line. Edges represent simulation and relativized simulation. Proposition 12 explains how to interpret transitivity in the hierarchy of systems (Fig. A.18).

Proposition 12 *For any systems $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$ and any two compatible color-blind environments $\mathcal{E}$ and $\mathcal{F}$ it holds that:* $\mathcal{E} \models \mathcal{S}_1 \leqslant \mathcal{S}_2 \wedge \mathcal{F} \models \mathcal{S}_2 \leqslant \mathcal{S}_3 \wedge \mathcal{E} \leqslant \mathcal{F} \Rightarrow \mathcal{E} \models \mathcal{S}_1 \leqslant \mathcal{S}_3$.

8 Model Transformations

While discussing techniques for implementation of optimizers, we rely as much as possible on existing techniques that are well known in compiler technology, partial evaluation and model-checking. What is important, we believe that optimizations should be designed and implemented for particular languages, not for their abstract semantics. Despite the fact that many transformations can be generalized beyond the particular language, such generalizations rarely lead to successful implementations in specialization tools and compilers. There is very little hope that we can translate any reactive synchronous program into an IOATS, apply the transformations on the abstract layer, and then transform it back to the modeling language, code generate and still obtain significantly optimized code. Contrary—it is most likely that this process of multiple translation will increase the code size significantly. Instead the semantic layer of IOATS should be used for making proofs of correctness of transformations operating on the level of concrete modeling language. We also use the semantic properties, like color-blindness, to inspire our search for transformations.

8.1 Environment Independent Transformations

Many simple optimizations can be achieved just by analysis of the state/event models in absence of any environment, relying on classical data-flow analysis [App98, chpt. 17] [Muc97, chpt. 8] and interpreting the model as a control graph. These include constant propagation [Muc97, p. 329], deadcode elimination [Muc97, p. 580], and, to some extent, elimination of side-effect-free code [Muc97, p. 592]. In reactive synchronous models, like state/event systems, these optimizations correspond to simplification of guards, dropping transitions never enabled, dropping states not being targets of any transitions, and dropping entire state machines or processes that produce no side-effects and are never referred to. The notion of side-effect-free part of the model is perhaps a bit less standard than the others, so let us discuss it briefly.

A state machine is *pure* if there are no guards in other machines that refer to it, and it has no outputs on transitions (this is a static property). A set of state machines is *mutually pure* if none of the state machines in the set has any outputs on transitions and none of the state machines outside the set refer to its states in guards. It is not hard to see that a maximal set of mutally pure state machines can safely be removed from the model without *any* execution environment ever observing this change. Formally this can be argued by making a two-way relativized simulation proof in our framework, assuming a perfect vision environment. Note that we have removed the sensor component from the alarm clock model using precisely this reasoning (see $\mathcal{C}_4$, Fig. A.7).

Typically, all optimizations mentioned above are not applicable for reactive models when used in isolation: programmers rarely write code that is dead, does

not compute anything or makes a complex computation that always results in the same constant. In most of such cases these are manifestations of errors, which should be reported. Basic optimizations are nevertheless essential, when combined with more advanced techniques described below, which often produce code with constants in expressions, unreachable states and transitions, etc.

8.2 Optimizations with Static Environments

Knowledge of static properties of environment may allow to apply the above optimizations more aggressively. If some events can never be input to the system, then they can obviously invalidate some guards (a never constraint on an event can be propagated as false to guard conditions and constant propagation may be applied subsequently). If some outputs are never being observed (always ignored), they can be erased from transitions' actions, which may introduce mutually pure components in the model, that could be ereased subsequently as described above.

Perhaps the most interesting optimizations, which are also new, can be performed for static environments exhibiting color-blindness (*Equiv* constraints). If some actions are always equivalent, we can definitely substitute one for another. Remember that actions correspond to invocations of actuator drivers, so if we could eliminate one driver/handler it can potentially save a significant amount of program memory. In order to justify the choice, one needs more information about the properties of drivers. In the following we assume that each action $a_i \in \mathit{Action}$ has a weight w_i assigned describing a measure to be minimized (execution time, code size, price of the respective actuator, etc). We formulate the problem as follows:

Input: a set of actions $\mathit{Action} = \{a_1, \ldots, a_n\}$, corresponding weights $w_1, \ldots, w_n$, and an equivalence relation E on *Action*.

Output: An optimal selection of equivalence class representatives, so that when substituted in the model for all call places they minimize the total cost for the model:

$$\sum_{i=1}^{n} x_i w_i \ ,$$

where $x_i = 1$ if a_i has been chosen as a representative for some class, and $x_i = 0$ otherwise. This problem can be solved computing equivalence classes using a Union-Find like algorithm, and then choosing the cheapest element for each class. Again proofs of correctness of the above optimizations can be written using two-way relativized simulation as a requirement, assuming an IOATS environment that represents the static properties required (all such environments have exactly one generator state and one observer state).

8.3 Optimizations with Dynamic Environments

There is no doubt, that introducing behavior in environments, vastly increases the amount of possible optimizations. For example precise reachability analysis can be executed on the model, to achieve more aggressive kind of code elimination, or identify more parts of guards as being constants. Nevertheless we shall not discuss all these possibilities here. Instead, we focus on a transformation that directly exploits color-blindness of our environments. Ultimately we would like to consider an algorithm for Action-Set-Minimization, an extension of the one presented in the previous paragraph, but with the equivalence relation on actions changing dynamically. This complicates the problem significantly, as now each transition, not each action, may require choosing a different substitute for its actions. Moreover this substitute candidate has to satisfy observation requirements of all environment transitions that can possibly be executed in parallel with the given system transition.

We can use reachability analysis [LNAH$^+$01, EMCGP99] to find pairs of environment/system transitions that can fire simultaneously. We are particularly interested in identifying pairs containing one observation transition of environment and one generation transition of the system. During the model exploration one should record information about what sets of actions are observed in what observation classes of environments, ultimately gathering a list of pairs: each consisting a subset of call places and a formula describing the observation class. This information can be translated into a constraint satisfaction problem, with the objective to minimize a cost function (most typically reflecting the code size as above). An off-the-shelf tool like ILOG's CPLEX [ILO] could be used to solve the problem, returning for each set of actions a cheaper replacement for this set. In fact, as it is typical in industrial optimization problems, the optimality of the solution is not required for the correctness of our application, so an approximating solver could be used (for example LP (Linear Programming) based), or the ILP (Integer Linear Programming) solver can be interrupted early to use the best solution found so far, if finding the optimal solution proves intractable. Finally the information obtained from the solver is used to restructure the transition layout and actual calls placed on transitions. Ultimately less drivers are needed in the final product, and the kernel code is smaller.

9 Related Work

Derivation of product lines is often associated with partial evaluation [JGS93, DGT96, HMT99]. There have been approaches to enable partial evaluation based on execution traces instead of fixed input values [HKY96, Mur95, EG98], nevertheless they were never implemented for realistic languages. We fear that these transformations, designed for abstract process calculi, can be barely applied in

such contexts. Our framework allows more transformations than known before due to the color-blindness, which allows some non-reductive mutations in the program.

Wąsowski [Wąs04] presented a static framework for specifying environments for reactive models, which relies solely on state independent properties. The present paper, itself an extension of [LLW05], provides a theoretical foundation for a product line management setup similar to Wąsowski's [Wąs04], but based on behavioral properties.

Relativized simulation has been introduced by Larsen in [Lar87, Lar86, LM92]. Our framework is modeled after this work, rephrased in the setting of IOATSs and extended with color-blindness. In Larsen's formulation, based on simple labeled transition systems [Mil89], it was impossible to directly express an environments' inability to distinguish outputs. Further results on color-blind environments, as well as detailed proofs, can be found in [Wąs05].

The study of systems embedded into behavioral contexts is quite mature [Lar86, AH01, Lyn88, RR02, IK01]. Our work stems out from the field, by its direct support for observability specifications via color-blindness. This support is needed, if the tools based on this framework, are to be useful for development of product lines of embedded systems.

Czarnecki and Antkiewicz [CA05] present a dual approach to product line derivation. We specify variants by legal use (black-box), they specify variants by annotating model internals with conditions (white-box). In our case the derivation is difficult, but safety properties are preserved. In their case the derivation is relatively easier, while specifications still need to be verified.

10 Conclusion & Future Work

We have presented the semantics of a specification language for environments of reactive synchronous systems, together with a notion of context-dependent refinement based on color-blindness. This refinement relation is more liberal than usual in allowing some mutations to program outputs, instead of bare reductions. We have explained and demonstrated how partial specifications of behaviors can be composed and used to define families of products. The framework was designed as a core of an upcoming tool for compact code generation and product line derivation for discrete control embedded systems. Our specifications shall be used as preconditions for advanced model optimizers/specializers. We have thoroughly discussed issues, which arise in the implementation of the theory for realistic languages, especially focusing on languages with sequences as outputs.

An implementation [sco] of a powerful context-aware optimizer for models based on model-checking and program analysis is planned. This prototype tool is supposed to be compatible with an industrial development environment for embedded systems [IAR], which will allow for realistic case studies.

Paper B

Interface Input/Output Automata

Kim G. Larsen, Ulrik Nyman
Department of Computer Science,
Aalborg University, Denmark

Andrzej Wąsowski
Computational Logic and Algorithms Group,
IT University of Copenhagen, Denmark

Abstract

Building on the theory of interface automata by de Alfaro and Henzinger we design an interface language for Lynch's I/O automata, a popular formalism used in the development of distributed asynchronous systems, not addressed by previous interface research. We introduce an explicit separation of assumptions from guarantees not yet seen in other behavioral interface theories. Moreover we derive the composition operator systematically and formally, guaranteeing that the resulting compositions are always the weakest in the sense of assumptions, and the strongest in the sense of guarantees. We also present a method for solving systems of relativized behavioral inequalities as used in our setup and draw a formal correspondence between our work and interface automata.

Keywords: I/O automata, Interface Theory, Interfaces, Behavioral Inequalities

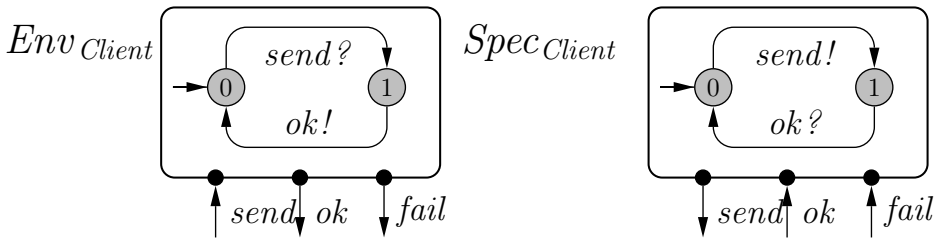


Figure B.1: $Client = (Env_{Client}, Spec_{Client})$

1 Introduction

A suitably expressive interface language lies at the very center of any component-oriented development framework. Interfaces are abstractions of components, carrying all essential information necessary to establish cross-component compatibility. Instead of reasoning about components directly, one typically examines compatibility of their interfaces, while the adherence of a particular implementation to its interface is tested separately. This, not only allows for independent development of components, but also by introducing compositionality helps to combat the state space explosion problem in various automatic analyses.

Type annotations, type checking, and type inference have traditionally been used to decide compatibility of components soundly with respect to memory safety. However, static type correctness in this traditional sense fails to guarantee more elaborate properties, like correctness of communication, or deadlock freeness. This observation has inspired a long line of research on behavioral type systems and behavioral interface languages suitable for specification of highly trusted computer systems (see [IK01, RR02, LX04, LZZ05] and references therein for examples).

We follow de Alfaro and Henzinger [AH01, AH04] in studying an automata based interface language, or *interface automata*. Unlike them however, we explicitly separate, in the interface description, the assumptions that a component may make about its use from the guarantees that it needs to commit to. Assumptions describe the possible behaviors of the component's external environment, while guarantees describe the possible behaviors of the component itself.

Each interface in our theory consists of two I/O automata. The first, called the *environment*, represents assumptions. The second, called the *specification*, describes guarantees. Figure C.1 shows an interface for a *Client* component consisting of the automata Env_{Client} and $Spec_{Client}$. The arrows incoming to or outgoing from the box surrounding each of the automata visualize their static types, or *signatures*. The environment Env_{Client} specifies that even though the static type does allow a *fail* action, the emission of this action is disallowed for all compliant execution environments. The only legal input is *send*. One can still use the *Client* component in a context that syntactically permits *fail*, but the

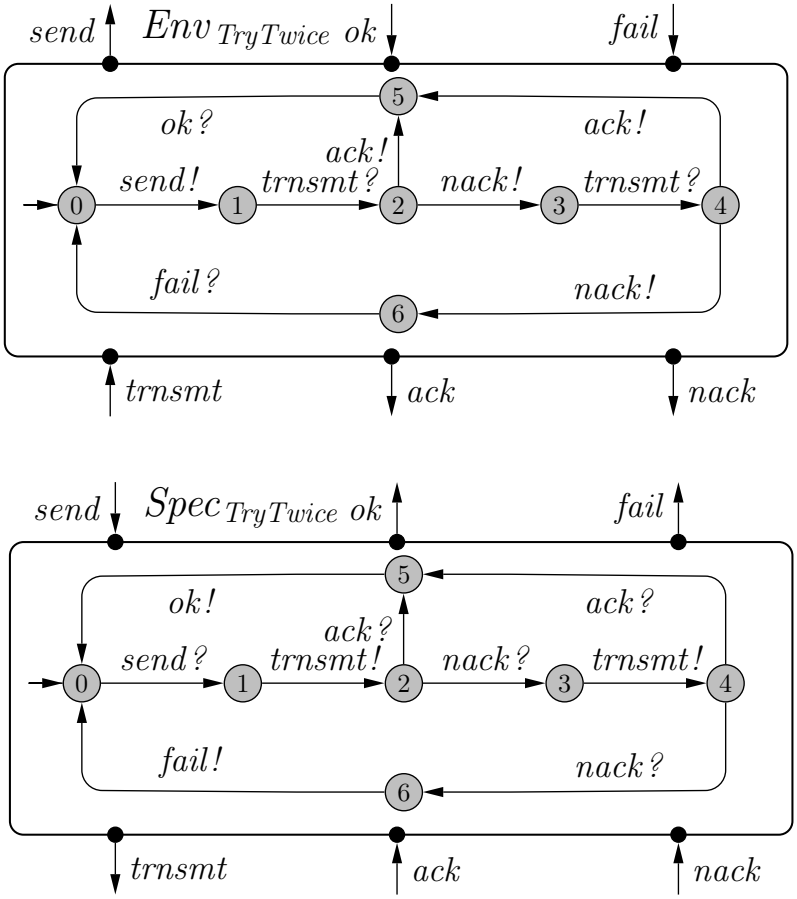


Figure B.2: $TryTwice = (Env_{TryTwice}, Spec_{TryTwice})$

behavior of the *Client* is only guaranteed in environments that do not fail.

Alfaro and Henzinger model assumptions about the use of a component by the interface's inabilities to receive inputs. The output transitions of the very same interface automaton describe its guarantees. Since we separate the two, we alleviate the need for blocking. Our automata are *input enabled*—accepting any input from their signature in every state. In order to avoid clutter we usually do not draw loop transitions, which correspond to ignoring an input. There is one such implicit transition $1 \xrightarrow{send?} 1$ in Env_{Client} and three in $Spec_{Client}$.

Two interfaces can be combined into a composite interface, describing a new set of assumptions and guarantees. Interface *TryTwice*, presented in Fig. B.2 can be composed with *Client*. The two components do not form a closed system, but are intended for use together with a further unspecified *LinkLayer* component.

Composition of interfaces is a central construction in any interface theory. One of our contributions is that the composition is derived systematically: we formally state requirements for it in the form of a system of inequalities, and derive a result of the composition as a maximal solution of this system. Consequently properties

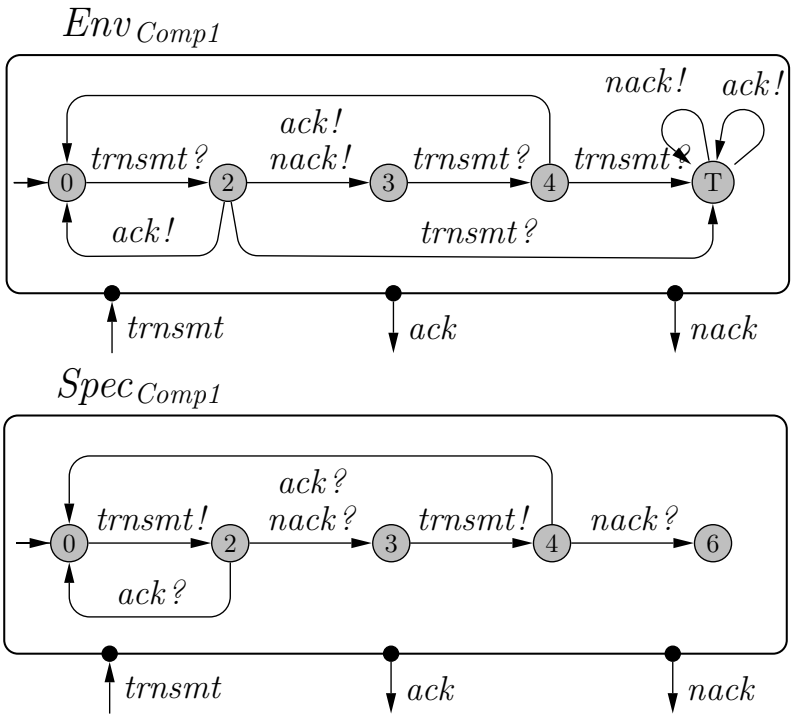


Figure B.3: $(Env_{TryTwice}, Spec_{TryTwice}) \otimes (Env_{Client}, Spec_{Client}) = Comp1$

of the composition hold by construction.

Figure B.3 shows the interface resulting from composing *Client* and *TryTwice*. Later we shall explain how it has been computed. Now observe that any component legally interacting with this new interface may not send a *nack* twice in response to the *transmt* request—a simple consequence of the fact that this would make *TryTwice* respond with a *fail* to *Client*, violating the assumptions of the latter. The additional state T manifests the fact that the computed environment expresses the weakest assumptions. It allows receiving arbitrary behavior after a second *transmt* in a row, because any compliant implementation would never send it, and thus would never be affected by the subsequent behaviour.

An advantage of separating assumptions from guarantees is that one of the automata can be changed without affecting the other. Thus the same guarantees can be used for multiple interfaces. In [LLW05] we have argued that this is useful for modeling software product lines: a family of component variants may be specified using a single specification (guarantee) and multiple environmental restrictions (assumptions). An advanced compiler may use the assumptions to derive specialized versions of the component from the same source code. Let us illustrate this with an example. Figure B.4a gives an alternative environment Env_{NoNack} for the $Spec_{TryTwice}$ specification. This environment disallows the sending of a *nack* as a response to a *trnsmt* request. Any implementation of *TryTwice* is also an implementation of $(Env_{NoNack}, Spec_{TryTwice})$. If it is only used in Env_{NoNack}, then it could be automatically specialized to these specific circumstances. The error

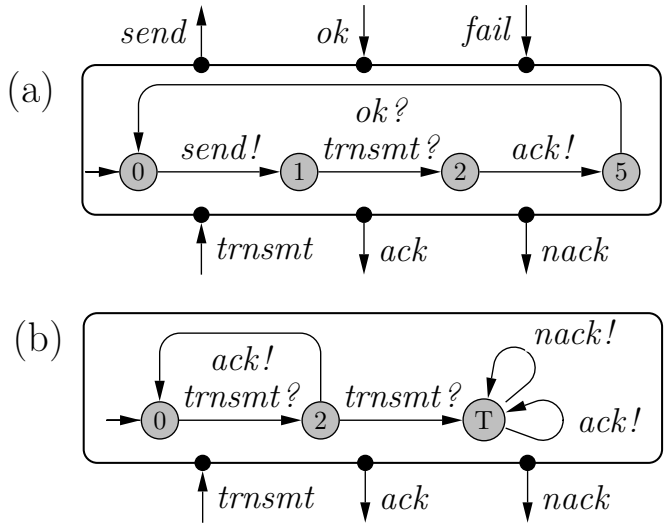


Figure B.4: (a) The environment Env_{NoNack} and (b) the environment Env_{Comp2}.

handling code could be removed as it is not needed in such a context. The composition $Comp2 = (Env_{NoNack}, Spec_{TryTwice}) \otimes (Env_{Client}, Spec_{Client})$ has exactly the same specification part as the *Comp1* composition. The resulting environment Env_{Comp2} (Fig. B.4b) disallows the generation of the *nack* input even though the static type permits this.

As we have also argued in [LLW05] the separation supports a simple declarative style of modeling assumptions: simple properties can be modeled as standalone automata and combined using the process algebraic operators of sum and product, corresponding to disjunction and conjunction of properties respectively.

An interesting theoretical side effect of our exposition, is an informal correspondence drawn between blocking and non-blocking interface theories. A single blocking interface automaton of [AH01] expresses both the assumptions of a component and its commitments. When a blocking interface automaton is unable to accept an input, it effectively assumes that any compatible environment will never provide it. In the theory for non-blocking systems the interfaces are composed of two non-blocking automata, and the same effect is achieved by explicitly using one of the automata for describing the permissible behavior of the surroundings.

The paper develops as follows. Section 2 defines I/O automata and interfaces. Section 3 discusses refinement of interfaces. The most central section, Section 4, is devoted to composition, while a more technical section, Section 5, is devoted to systems of inequalities used in section 4 and is a contribution in itself. But reading it is not essential for appreciating our interface theory. Section 6 draws a correspondence between interface automata and our interfaces, while section 7 discusses other related work. We conclude in section 8. A particularly interested reader can find the proofs of all our claims in an upcoming BRICS report.

2 I/O Automata and Their Interfaces

Definition 1 *An I/O automaton* $S{=}(states_S, start_S, in_S, out_S, int_S, steps_S)$ *is a 6-tuple, where* $states_S$ *is a set of states,* $start_S \in states_S$ *is an initial state,* in_S *is a set of input actions,* out_S *a set of output actions, and* int_S *is a set of internal actions. All of the action sets are mutually disjoint. We abbreviate* $ext_S = in_S \cup out_S$ *and* $act_S = ext_S \cup int_S$*. Then* $steps_S \subseteq states_S \times act_S \times states_S$ *is the set of transitions. I/O automata are input enabled: for every state* s *and any action* $i \in in_S$ *there exists a state* s' *and a transition* $(s, i, s') \in steps_S$*.*

We write $q \xrightarrow{a}_S q'$ if $(q, a, q') \in steps_S$. We often explicitly suffix external actions with direction of communication writing $q \xrightarrow{a!}_S q'$ if $a \in out_S$, and $q \xrightarrow{a?}_S q'$ if $a \in in_S$. Notice that the labels $a!$ and $a?$ still denote exactly the same action, and we can drop the suffixes whenever the direction of communication is irrelevant. We write $q \xrightarrow{a}\!\!\!\!\!\!/\;\;$, meaning that there is no q' such that $q \xrightarrow{a} q'$.

Definition 2 *An execution of an I/O-automaton* S *starting in a state* q^0 *is a finite sequence of labels* $q^0, a_0, q^1, a_1, q^2, a_2, \ldots, q^{n-1}, a_{n-1}, q^n$ *such that all* q^i*'s are members of* $states_S$*, all* a_i*'s are members of* act_S *and for every* $k = 0 \ldots n-1$ *it is the case that* $q^k \xrightarrow{a_k}_S q^{k+1}$*. A trace* σ *of* S *is an execution* ψ *of* S *starting in the initial state, with all the states and internal actions deleted:* $\sigma = \psi \restriction ext_S$*, where* $\psi \restriction X$ *denotes a sequence created from* ψ *by removing symbols that are not in set* X*. The set of all traces of automaton* S *is denoted* Tr_S*.*

Two I/O-automata S_1 and S_2 are *syntactically composable* if their input and output sets do not overlap and their internal actions are not shared: $in_{S_1} \cap in_{S_2} = out_{S_1} \cap out_{S_2} = int_{S_1} \cap act_{S_2} = act_{S_1} \cap int_{S_2} = \emptyset$. Two syntactically composable automata $S_1{=}(states_{S_1}, start_{S_1}, in_{S_1}, out_{S_1}, int_{S_1}, steps_{S_1})$ and $S_2{=}(states_{S_2}, start_{S_2}, in_{S_2}, out_{S_2}, int_{S_2}, steps_{S_2})$ can be composed into a single product automaton $S = S_1 \otimes S_2$, where $S{=}(states_S, start_S, in_S, out_S, int_S, steps_S)$ and $states_S = states_{S_1} \times states_{S_2}$, $start_S = (start_{S_1}, start_{S_2})$, $in_S = in_{S_1} \cup in_{S_2} \setminus out_{S_1} \setminus out_{S_2}$, $out_S = out_{S_1} \cup out_{S_2} \setminus in_{S_1} \setminus in_{S_2}$, $int_S = int_{S_1} \cup int_{S_2} \cup (ext_{S_1} \cap ext_{S_2})$, and $steps_S$ are defined by the following rules:

$$\text{if } q_1 \xrightarrow{a}_{S_1} q_1' \text{ and } a \in act_{S_1} \setminus act_{S_2} \text{ then } (q_1, q_2) \xrightarrow{a}_{S_1 \otimes S_2} (q_1', q_2)$$
$$\text{if } q_2 \xrightarrow{a}_{S_2} q_2' \text{ and } a \in act_{S_2} \setminus act_{S_1} \text{ then } (q_1, q_2) \xrightarrow{a}_{S_1 \otimes S_2} (q_1, q_2')$$
$$\text{if } q_1 \xrightarrow{a}_{S_1} q_1' \text{ and } q_2 \xrightarrow{a}_{S_2} q_2' \text{ then } (q_1, q_2) \xrightarrow{a}_{S_1 \otimes S_2} (q_1', q_2')$$

In practice unreachable states may be removed from the product, without affecting the results presented below.

Our composition (same as in [AH04]) differs from the standard I/O automata composition in that it applies hiding immediately. It is equivalent with the standard composition as long as each action is only shared by at most two components.

We define an interface model to be a pair (E, S) of I/O automata:

Definition 3 *A pair of I/O automata (E, S) is an interface if $E \otimes S$ is a closed system, i.e. $in_E = out_S$ and $out_E = in_S$.*

The environment automaton E drives the specification automaton S. Any implementation I of S must conform to S as long as it is receiving input that conforms to E. The behavior of I on sequences of inputs that cannot be provided by E is not constrained. We formalize this using relativized refinement:

Definition 4 *An I/O automaton I implements an interface (E, S), written $E \models I \leqslant S$, iff $out_I = out_S$ and $in_I = in_S$ and $Tr_E \cap Tr_I \subseteq Tr_S$.*

3 Refinement of Interfaces

We establish a hierarchy on interfaces in order to quantify their generality.

Definition 5 *Let (E_1, S_1) and (E_2, S_2) be two interfaces with the same signatures. We will say that (E_1, S_1) is a stronger interface than (E_2, S_2), written $(E_1, S_1) \preceq (E_2, S_2)$, if (E_1, S_1) has less implementations than (E_2, S_2), so for any I/O automaton I: $E_1 \models I \leqslant S_1$ implies $E_2 \models I \leqslant S_2$.*

The refinement of interfaces can be seen as a subtyping relation in a behavioral type system for components. In such an interpretation we would say that (E_1, S_1) is a subtype of (E_2, S_2). We propose several simple sound characterizations of the above refinement that are useful in making proofs:

Theorem 1 *Let (E_1, S_1), (E_2, S_2) be interfaces with identical signatures. Then*

1° *$Tr_{E_1} \cap Tr_{S_1} = Tr_{E_2} \cap Tr_{S_2}$ implies $(E_1, S_1) \preceq (E_2, S_2)$ and $(E_2, S_2) \preceq (E_1, S_1)$*

2° *$Tr_{E_2} \subseteq Tr_{E_1} \wedge Tr_{S_1} \subseteq Tr_{S_2}$ implies $(E_1, S_1) \preceq (E_2, S_2)$*

3° *$Tr_{E_1} \setminus Tr_{S_1} \supseteq Tr_{E_2} \setminus Tr_{S_2}$ implies $(E_1, S_1) \preceq (E_2, S_2)$*

The above characterizations are convenient in establishing subtyping relations among interfaces in many concrete cases. However none of them are complete. The refinement of interfaces can be characterized in a sound and complete manner using a notion of tests that resembles failure traces of Hoare [Hoa85], but determinized, relativized with respect to the environment, and suffix closed.

Definition 6 *The set of conformance tests of interface (E, S) is defined as:*

$$test_{(E,S)} = \{\sigma \cdot a \mid \sigma \in Tr_E \cap Tr_S,\ \sigma \cdot a \in Tr_E \setminus Tr_S\} \cdot ext_E^* \ ,$$

where X^* denotes the set of all finite sequences over alphabet X.

Theorem 2 *Let (E_1, S_1) and (E_2, S_2) be two interfaces with identical signatures. Then $test_{(E_1,S_1)} \supseteq test_{(E_2,S_2)}$ iff $(E_1, S_1) \preceq (E_2, S_2)$.*

Without spelling out the details, we remark that a finite automaton, such that $test_{(E,S)}$ is its accepted language, can be computed in quadratic time, and can be used for testing containment in applications of the above theorem.

4 Interface Compositions

We would like to abstract compositions of components by compositions of their interfaces. For any two compatible interfaces (E_1, S_1) and (E_2, S_2) we should be able to derive an interface of their composition (E, S), the one that is implemented flawlessly by any two implementations of (E_1, S_1) and (E_2, S_2).

Two interfaces are *syntactically composable* if the I/O automata comprising them are pointwise syntactically composable. This guarantees that any components I_1 and I_2 implementing syntactically composable interfaces (E_1, S_1) and (E_2, S_2), are also syntactically composable. The question that we want to address is the *dynamic compatibility* of I_1 and I_2: can I_1 violate the environmental assumptions expressed in E_2? Can I_2 violate the assumptions in E_1?

We may be tempted to say that the composite interface is the composition of the interface parts: $(E, S) = (E_1 \otimes E_2, S_1 \otimes S_2)$. This construction, however, is unsound. It is possible to find two compliant implementations that, when composed together, violate (E, S). In order to arrive at a sound and complete notion of composition, we will state the requirements for the composite interface, and then derive the construction from them. The three requirements are: *independent implementability* [AH04], *mutal deadlock freeness*, and *associativity*.

Independent implementability means that (E, S) is such, that the implementations of (E_1, S_1) and (E_2, S_2) can be developed independently of each other, and their composition will implement the composition of their interfaces:

$$\text{For all } I_1, I_2.\ E_1 \models I_1 \leqslant S_1 \text{ and } E_2 \models I_2 \leqslant S_2 \text{ implies } E \models I_1 | I_2 \leqslant S\ . \quad \text{(B.1)}$$

Mutual deadlock freeness means that any two correct implementations, when composed and embedded in an environment that obeys the assumptions of E, will not violate each other's assumptions:

$$\begin{aligned}\text{For all } I_1, I_2. E_1 \models I_1 \leqslant S_1 \text{ and } E_2 \models I_2 \leqslant S_2 \\ \text{implies } I_1 \models E \otimes I_2 \leqslant E_1 \text{ and } I_2 \models E \otimes I_1 \leqslant E_2\ .\end{aligned} \quad \text{(B.2)}$$

You may find it useful to refer to the flowgraph on Fig. B.5a, while studying the above rule. Observe that in the composed system I_1 is indeed the environment in which $E \otimes I_2$ operates. The composition $E \otimes I_2$ is also the environment for I_1 and it is supposed not to violate any of the assumptions expressed in E_1.

Finally, associativity means that in whatever order compositions are applied, they give rise to equivalent interfaces:

$$\begin{aligned}((E_1, S_1) \otimes (E_2, S_2)) \otimes (E_3, S_3) \preceq (E_1, S_1) \otimes ((E_2, S_2) \otimes (E_3, S_3)) \\ (E_1, S_1) \otimes ((E_2, S_2) \otimes (E_3, S_3)) \preceq ((E_1, S_1) \otimes (E_2, S_2)) \otimes (E_3, S_3)\ .\end{aligned} \quad \text{(B.3)}$$

A disadvantage of the above requirements is that they are not constructive. They rely on quantification over all implementations, which makes them useless

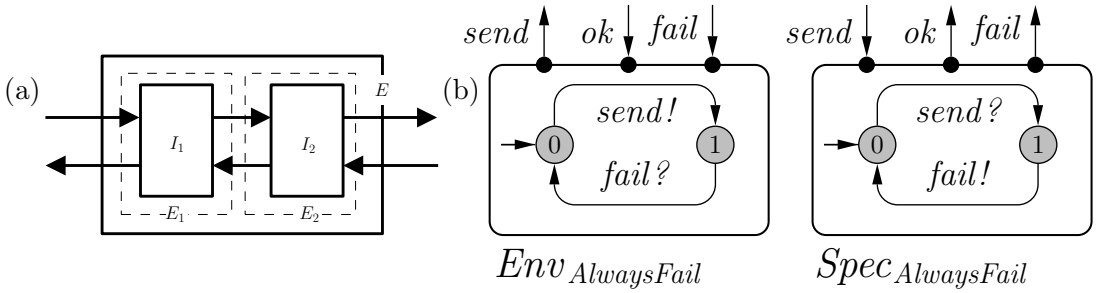


Figure B.5: (a) Flowgraph for a composition of (E_1, S_1) and (E_2, S_2). (b) *AlwaysFail*

for computing the composition. Fortunately the quantification can be eliminated. The following theorem reduces the property of mutual deadlock freeness of all implementations to mutual deadlock freeness of the interfaces being composed:

Theorem 3 *Any environment E fulfills the requirement (B.2) iff it fulfills the following condition:*

$$S_1 \models E \otimes S_2 \leqslant E_1 \text{ and } S_2 \models E \otimes S_1 \leqslant E_2 \ . \tag{B.4}$$

The above reduction is very fortunate, as (B.4) also implies independent implementability with the choice of the guarantees component to be $S_1 \otimes S_2$:

Theorem 4 *Let (E_1, S_1) and (E_2, S_2) be syntactically composable interfaces, and E be an environment I/O automaton satisfying property (B.4). Then for all I_1 and I_2 such that $E_1 \models I_1 \leqslant S_1$ and $E_2 \models I_2 \leqslant S_2$ we have $E \models I_1 \otimes I_2 \leqslant S_1 \otimes S_2$.*

Consequently if we were able to find an environment E satisfying (B.4), then the interface $(E, S_1 \otimes S_2)$ would satisfy mutual deadlock freeness and independent implementability—a good candidate for the composition of environments. However, the environment satisfying (B.4) may not always exist. This is the case, if S_1 unconditionally, independently of E's behavior, violates the assumptions of S_2 expressed in E_2. In this case (E_1, S_1) and (E_2, S_2) are said to be *incompatible*.

Definition 7 *Interfaces (E_1, S_1), (E_2, S_2) are incompatible if there exists no I/O automaton E such that: $S_1 \models E|S_2 \leqslant E_1$ and $S_2 \models E|S_1 \leqslant E_2$.*

Figure B.5b shows an interface *AlwaysFail*, which has a signature compatible with the signature of *Client*. Nevertheless the dynamic types of *Client* and *AlwaysFail* are incompatible in that they share only one nonempty trace, consisting of one step, and this trace ends in a deadlock.

In fact there typically exist many pairs (E, S) that satisfy all our requirements. For example an interface (M, U), consisting of a mute environment M never producing any outputs and a universal system specification U generating

all possible traces, would satisfy the composition requirements of any two compatible interfaces. The interface (M, U) allows any implementation—it says that its implementations will behave in an arbitrary fashion (U), not allowing any external stimulation (M). Clearly, as a component interface, (M, U) is useless.

We should ensure that our composition operator produces the interface that carries over all the information available from its components. It must have the smallest possible set of implementations, while still satisfying all our requirements. Similarly, it must maximize the set of components compatible with it (as opposed to the set of components implementing it). We shall call this optimal interface *the most general*. Intuitively to achieve this optimality we need an environment E satisfying the requirements such that it is maximal with respect to trace inclusion. By increasing the set Tr_E we make it easier for components to be compatible with our interface. Similarly we make it harder to implement the composite interface, as increasing the set of traces of E decreases the assumptions that an implementation can make. The following theorem says that such a maximal E always exists for compatible interfaces:

Theorem 5 *Let (E_1, S_1) and (E_2, S_2) be two syntactically composable interfaces. If there exists an I/O automaton E enjoying property (B.4) then there also exists a maximal such environment with respect to trace inclusion.*

Theorem 6 *The composition operator mapping interfaces (E_1, S_1) and (E_2, S_2) to $(E, S_1 \otimes S_2)$, where E is the maximal solution of (B.4), is associative.*

Theorems 5–7 together with our earlier observations suggest that the interface $(E, S_1|S_2)$, where E is this maximal solution of equations (B.4), is even more likely to be the most general interface that we are searching for. A maximal solution of (B.4) can be found algorithmically for finite state interfaces. Section 5 describes a method that can be used for this purpose.

As increasing the environment E makes the interfaces more general, so does decreasing the specification S (within the limits set by the requirements). For any particular selection of E satisfying (B.1), no S can be smaller (relative to E) than $S_1|S_2$, because S_1 and S_2 themselves are valid implementations. So $S_1|S_2$ is the smallest possible specification of the composite interface with respect to any particular choice of E. This observation can be generalized to a claim that $(E, S_1 \otimes S_2)$ is the most general interface possible:

Theorem 7 *Let (E_1, S_1), (E_2, S_2) be interfaces. Let E be the maximal solution to (B.4) and let (E', S') satisfy independent implementability and mutual deadlock freeness. If (E', S') is compatible with (E'', S'') then also $(E, S_1 \otimes S_2)$ is compatible with (E'', S'').*

Having concluded that $(E, S_1 \otimes S_2)$, where E is a maximal solution of (B.4), is well defined and the most general, we can use it as a definition of the composition operator. We will denote this composite interface by $(E_1, S_1) \otimes (E_2, S_2)$.

Furthermore our composition of interfaces is complete in the following sense

Theorem 8 *For compatible interfaces* (E_1, S_1), (E_2, S_2) *and any* (E', S') *satisfying independent implementability and mutual deadlock freeness:*

$$(E_1, S_1) \otimes (E_2, S_2) \preceq (E', S') \ .$$

We remark that our composition would not be complete if we only required independent implementability. It seems likely from the work presented in [Mai03] that it is indeed impossible, for our setting, to be complete in the above sense using only independent implementability. Similarly we would not be complete if we only required mutual deadlock freeness, simply because it does not restrict the S component, which can then be taken to be mute, likely yielding a smaller interface than ours. Still our composition is sound and complete with respect to both requirements combined. Requirements (B.2) and (B.3) have been introduced solely for their inherent usefulness. Their interplay guaranteeing soundness and completeness is a pleasant side effect.

Definition 8 *Let* (E_1, S_1), (E_2, S_2) *be syntactically composable interfaces. Their composition, denoted* $(E_1, S_1) \otimes (E_2, S_2)$, *is an interface* $(E, S_1 \otimes S_2)$, *where* E *has the same signature as* $E_1 \otimes E_2$, *and is a maximal solution of (B.4).*

The operator of Def. 8 is associative, supports independent implementability and mutual deadlock freeness, and produces the most general interfaces.

5 Solving Behavioral Inequalities

Computing compositions of interfaces requires a method for finding solutions of systems of relativized linear inequalities. In particular we are interested in systems of inequalities of the following form:

$$\mathcal{C}(E) \ : \ \begin{cases} P_1 \models E \otimes S_1 \leqslant F_1 \\ \quad \vdots \\ P_m \models E \otimes S_m \leqslant F_m \end{cases} \tag{B.5}$$

where $\{P_i\}_{i=1..m}$, $\{S_i\}_{i=1..m}$ and $\{F_i\}_{i=1..m}$ are states of the three I/O automata P, S and F and E is a single unknown automaton. We are interested in finding a greatest such E with respect to $\leqslant$, or in reporting incompatibility between components, if no solutions exist. Since in (B.4) various components of inequalities come from separate automata, in order to apply the method below we need to construct three automata P, S and F as the disjoint unions of the automata that appear in the given place of the constraints in (B.4). We introduce three convenient mapping functions *in*, *out* and *ext* which from a state of the two automata F and S return respectively the set of input, output or external actions of the automata that this state originates from in the disjoint union computation. We

will use them in the algorithm below to recover some of the signature information lost by making the disjoint union.

For simplicity of exposition we shall also assume that all I/O automata involved in the systems are deterministic. Otherwise they can be determinized without loss of information, as long as our refinement criterion is based on language inclusion. This assumption is not inherent to the method, though.

We should now state a property similar to Theorem 5, but formulated for systems of inequalities in general. We expand it to any number of constraints and do not require that all the I/O automata come from the same interfaces.

Theorem 9 *Let $\mathcal{C}(E)$ be a finite system of relativized inequalities:*

$$\mathcal{C}(E) \;:\; \begin{cases} P_1 \models E \otimes S_1 \leqslant F_1 \\ \quad\vdots \\ P_m \models E \otimes S_m \leqslant F_m \end{cases}$$

If $\mathcal{C}(E)$ has a solution (an I/O automaton satisfying all the constraints), then $\mathcal{C}(E)$ also has a greatest solution with respect to trace set inclusion.

We begin with constructing a *modal transition system* [LT88] corresponding to $\mathcal{C}(E)$, and then choose a maximal solution from its states and transitions. From our perspective modal transition systems are automata with two transition relations $\rightarrow_{may}$ and $\rightarrow_{must}$.

Definition 9 *A modal transition system is a quadruple $\mathcal{S} = (Q, A, \rightarrow_{may}, \rightarrow_{must})$, where Q is a set of systems of constraints (states), A is a set of actions, $\rightarrow_{may} \subseteq Q \times A \times Q$ is the* may *transition relation, and $\rightarrow_{must} \subseteq Q \times A \times Q$ is the* must *transition relation, $\rightarrow_{must} \subseteq \rightarrow_{may}$.*

Systems of relativized inequalities can be seen as sets of constraint triples $\{(P_1, S_1, F_1), \ldots, (P_m, S_m, F_m)\}$ over the solution E. The constraints evolve when any of their components, including the unknown E, takes an action. This evolution comprises not only state changes of the I/O automata, but also removing and introducing constraints. Legal actions of the unknown component E in any of its states are dependent on the states of the constraints—on what all the P_i's, S_i's and all the F_i's can do. This is why we label states of our modal transition systems with systems of inequalities (sets of constraints). All the steps that are allowed by the constraints, but are not strictly required (like a possibility to produce an output) should give rise to *may* transitions in the modal transition system. While all the steps that are strictly required (like input actions enforced by input-enabledness) give rise to corresponding *must* transitions.

Formally three I/O automata P, S, F induce a modal transition system $\mathcal{E} = (Q, A_0, \rightarrow_{may}, \rightarrow_{must})$, where elements of Q are sets of constraints over states of P, S and F, enriched with a distinct primitive constraint FALSE denoting an

empty set of solutions. The initial state A_0 is equal to the set $\{(P_1, S_1, F_1), \ldots, (P_m, S_m, F_m)\}$ of initial constraints, and the transition relations are defined according to the following rules:

$E \xrightarrow{a!}_{may} E'$ if and only if both of the following rules are satisfied:

For all $(P, S, F) \in E$ such that $a \in out_E \setminus in_S$
If $\exists F'.\, F \xrightarrow{a!} F'$ and $\exists P'.\, P \xrightarrow{a} P'$ then $(P', S, F') \in E'$
Else if $\exists P'.P \xrightarrow{a?} P'$ and $F \xrightarrow{a!}\!\!\!\!\!\!/\;\;$ then FALSE $\in E'$

For all $(P, S, F) \in E$ and all S' such that $a \in out_E \cap in_S$
If $S \xrightarrow{a?} S'$ also $(P, S', F) \in E'$

$E \xrightarrow{a?}_{must} E'$ and $E \xrightarrow{a?}_{may} E'$ iff both of the following rules are satisfied:

For all $(P, S, F) \in E$ and all F' such that $a \in in_E \setminus out_S$
If $F \xrightarrow{a?} F'$ and $P \xrightarrow{a!} P'$ then $(P', S, F') \in E'$

For all $(P, S, F) \in E$ such that $a \in in_E \cap out_S$
If $S \xrightarrow{a!} S'$ then $(P, S', F) \in E'$

Each state $E \in Q$ of $\mathcal{E}$ is minimal such that it satisfies the above transition rules *and* the following *closure rules*:

For all $(P, S, F) \in E$ and $a \in ext_S \cap ext_F$
If $\exists S'.\, S \xrightarrow{a} S'$ and $\exists F'.\, F \xrightarrow{a} F'$ and $\exists P'.\, P \xrightarrow{a} P'$
then also $(P', S', F') \in E$.

For all $(P, S, F) \in E$ and $a \in ext_S \cap ext_F$
If $S \xrightarrow{a!} S'$ and $F \xrightarrow{a!}\!\!\!\!\!\!/\;\;$ and $\exists P'.\, P \xrightarrow{a?} P'$ then FALSE $\in E$.

The two *may* rules discuss E making an output transition concerning an external output, or an internal communication with S respectively. The *must* rules state that E needs to accept all the inputs from the outside and from S respectively. Finally the closure rules allow S to advance without any interference with E on its own external actions. Whenever there is a possibility of violation of the relativized trace inclusion, we add false to the target state of E, hinting that E should not be allowed to make that step.

Definition 10 *The state consistency relation $\mathcal{S}$ over a modal transition system $\mathcal{E} = (Q, A, \rightarrow_{may}, \rightarrow_{must})$ is the maximal subset of Q such that if $E \in \mathcal{S}$ then* FALSE $\notin E$ *and whenever $E \xrightarrow{a}_{must} E'$ then $E' \in \mathcal{S}$.*

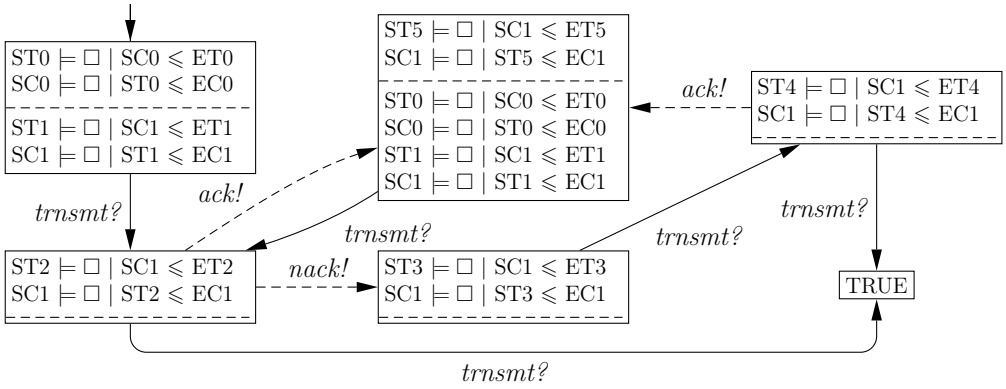


Figure B.6: The resulting modal transition system for the computation of Env_{Comp1}.

Definition 11 *A consistent set of transitions $\mathcal{T}$ of a modal transition system $\mathcal{E} = (Q, A, \rightarrow_{may}, \rightarrow_{must})$ with respect to consistency relation $\mathcal{S}$ is a maximal subset of $\rightarrow_{may}$, where whenever $(s, a, s') \in \mathcal{T}$ then $s \in \mathcal{S}$ and $s' \in \mathcal{S}$.*

Theorem 10 *Let $\mathcal{C}(E)$ be a system of inequalities as required above, and $\mathcal{E} = (Q, A, \rightarrow_{may}, \rightarrow_{must})$ be the modal transition system induced by $\mathcal{C}$. Then the maximal solution of $\mathcal{C}(E)$ is an I/O automaton E such that its set of states $states_E$ is a maximal consistency relation over $\mathcal{E}$,*

$$\begin{aligned} start_E &= \{(F_1, S_1), ..., (F_m, S_m)\}, \\ in_E &= \bigcup_{i=1}^{m}(in_{F_i} \setminus in_{S_i}) \cup \bigcup_{i=1}^{m}(out_{S_i} \setminus out_{F_i}) \\ out_E &= \bigcup_{i=1}^{m}(out_{F_i} \setminus out_{S_i}) \cup \bigcup_{i=1}^{m}(in_{S_i} \setminus in_{F_i}), \end{aligned}$$

and its set of transitions $step_E$ is a maximal consistent set of transitions of $\mathcal{E}$ with respect to $states_E$. If the maximal state consistency relation of $\mathcal{E}$ is empty then $\mathcal{C}$ has no solutions.

The set $\mathcal{S}$ can be found by a simple maximal fixpoint computation. In practice the consistency of the initial state may be decided in a local fashion without constructing the entire modal transition system.

Figure B.6 shows the consistent part of the modal transition system induced by $(Env_{TryTwice}, Spec_{TryTwice}) \otimes (Env_{Client}, Spec_{Client})$. It can then be minimized in order to obtain Env_{Comp1}, shown in Fig. B.3. Similarly $Spec_{Comp1}$ from Fig. B.3 has been obtained by minimizing $Spec_{TryTwice} \otimes Spec_{Client}$.

6 Interface Automata

The relation of our theory to interface automata [AH01, AH04] requires special attention, as we address several issues of that work; most importantly the representation of assumptions and guarantees within a single automaton. We clearly separate assumptions from guarantees, and the pairs of assumptions and guarantees can be constructed independently. In [AH04] Alfaro and Henzinger discuss static Assume/Guarantee interfaces featuring a similar split, however they do not persue the idea to the dynamic case.

In a larger perspective our work can be seen as a study of building interface theories as such: starting with a selection of the building blocks, going through requirements analysis, deriving the composition operator, and studying its generality. Let us review this process briefly. We begin with selecting important ingredients such as a component model, an interface model, an implementation relation and a refinement relation. The particular choice of input-enabled systems and (relativized) trace inclusion is not crucial for our developments. In fact we believe that a similar theory can be built using (relativized) simulation, or for timed automata. We choose I/O automata and trace inclusion because they are very different from Alfaro and Henzinger's interface automata, so we incidentally provide a component theory for a different community—the I/O automata community. At the same time our choice challenges some opinions expressed in [AH01, AH04] that building such a theory, especially supporting contravariant refinement, is impossible using language inclusion criteria or in a non-blocking setting.

Furthermore we show how the composition operator can be derived from requirements (by analysis, reduction and automated solving), while Alfaro and Henzinger introduce this operator in a rather ad hoc manner. After having derived our operator we discuss its generality, and conclude that it is indeed the most general operator possible, meeting our requirements with respect to trace inclusion, with respect to the $\preceq$ refinement, and with respect to compatibility with other components. We conjecture that the operator of our predecessors is also the most general in their setting, however they never make that claim.

Let us now draw a formal correspondance between the two interface theories.

Definition 12 (after [AH04]) *An interface automaton is a six-tuple* $S = (states_S, start_S, in_S, out_S, int_S, steps_S)$*, where* $states_S$ *is a finite set of states,* $start_S \in states_S$ *is an initial state,* in_S*,* out_S*, and* int_S *are three pairwise disjoint sets of input, output, and internal actions respectively, and* $steps_S \subseteq states_S \times act_S \times states_S$ *is an input-deterministic transition relation, with* $act_S = in_S \cup out_S \cup int_S$

Notice that the transition relation of interface automata may be non input-enabled. Syntactic composability of interface automata is governed by the same

rule as the composability of I/O automata, defined on p. 76. The composed interface is computed by taking a product of the two automata, and removing from it all *incompatible states*. A state of the product is an *error state* if one of its components can produce a shared output, that the other is unable to receive. A state of the product is *incompatible* if it can reach an error state by an execution over internally controllable transitions (transitions labeled with actions from: $int_{S_1 \otimes S_2} \cup out_{S_1 \otimes S_2}$).

Definition 13 *Two syntactically composable interface automata S_1 and S_2 are compatible iff removing all incompatible states from their product leaves an interface automaton with a non-empty set of reachable states.*

The function *unzip* defined below translates an interface automaton to an I/O automaton interface. If A is an interface automaton then $unzip_A := (E, S)$, where $states_S = states_E = states_A \cup \{T\}$, $start_S = start_E = start_A$, $in_S = out_E = in_A$, $out_S = in_E = out_A$, $int_S = int_E = int_A$. The transition relations of E and S are created from the transition relation of A by making it input-enabled on the respective input sets:

$$steps_E = steps_A \cup \{(s, a, T) | s \in states_A, a \in in_E, s \stackrel{a}{\not\rightarrow}_A\}$$

$$steps_S = steps_A \cup \{(s, a, T) | s \in states_A, a \in in_S, s \stackrel{a}{\not\rightarrow}_A\}$$

Theorem 11 *If A_1 and A_2 are two compatible interface automata, then $unzip_{A_1}$ and $unzip_{A_2}$ are compatible I/O automata interfaces.*

The *zip* function is a reverse of *unzip*: it translates an I/O automata interface into a single interface automaton, by computing the product of the two parts using the classic algorithm [HMU01, chpt. 4.2] from automata theory: $zip_{(E,S)} := A$, where $states_A = states_E \times states_S$, $start_A = (start_E, start_S)$, $in_A = in_S$, $out_A = out_S$, $int_A = int_S \cup int_E$, and $steps_A = \{((s, e), a, (s', e')) | s \stackrel{a}{\rightarrow} s' \text{ and } e \stackrel{a}{\rightarrow} e'\}$.

Theorem 12 *If (E_1, S_1), (E_2, S_2) are compatible deterministic I/O automata interfaces, then $zip_{(E_1,S_1)}$, $zip_{(E_2,S_2)}$ are compatible interface automata.*

The fact that our compatibility only implies compatibility in the interface automata sense for unzippings of deterministic interfaces is not surprising. It is actually expected, due to the very different nature of the refinement relations used in the two theories: trace inclusion and alternating simulation [AHKV98].

Alfaro and Henzinger choose alternating simulation to support contravariant treatment of inputs and outputs. We stress that input-enabledness and relativized trace inclusion already guarantee contravariant treatment of behaviors in a very similar spirit. Still our theory somewhat strictly requires that implementations of an interface have precisely the same sort as their interfaces, so it is technically not

possible to substitute a richer component in place of a simpler one, if they are the same on shared functionality. We stress that this deficiency is not inherent, while it simplifies the presentation. Contravariant signature extensions can be easily realized with relativized trace inclusion in the input-enabled setting. Instead of requiring $in_I = in_S$ and $out_I = out_S$ in Def. 3, insist on $in_S \subseteq in_I$ and $out_I \subseteq out_S$. In fact the only significant change required in later developments is the addition of a side condition to the independent implementability rule:

$$\begin{aligned}\forall I_1, I_2.E_1 \models I_1 \leqslant S_1 \text{ and } E_2 \models I_2 \leqslant S_2 \text{ and} \\ in_{I_1} \cap out_{S_2} \subseteq in_{S_1} \text{ and } in_{I_2} \cap out_{S_1} \subseteq in_{S_2} \text{ implies } E \models I_1|I_2 \leqslant S \ . \quad \text{(B.6)}\end{aligned}$$

This is the very same side condition that Alfaro and Henzinger add to independent implementability in order to support contravariant signature extensions. It ensures that even though the implementation allows additional inputs, it will only be used as described in this interface. The other components will not communicate with it on these additional inputs.

7 Other Related Work

Our work relates directly to the original version of interface automata [AH01, AH04], which was later extended with time and resource information in [AHS02] and [CdAHS03]. To strengthen the case, we have used some examples from [AH04] adapting them to our framework, and aligned the terminology with [AH01, AH04] as much as possible. Another approach to compatibility for blocking-services is taken by Rajamani and Rehof in [RR02] targeting compatibility of web services. We work in the input-enabled asynchronous setting of I/O-automata [Lyn88], which is semantically closer to implementations of embedded systems. To the best of our knowledge similar properties have not been studied in the I/O automata community yet.

The notion of relativized refinement and equivalence, or more precisely simulation and bisimulation, is due to Larsen [Lar86, Lar87]. It was so far applied in the setting of protocol verification [LM92], automatic testing [LMN05] and modeling software product lines [LLW05]. Here we adapt it to a language inclusion based refinement.

The general method of solving systems of behavioral equations using disjunctive modal transition systems and bisimulation as a requirement was published in [LX90]. The method presented in section 5 is an adaptation of this earlier work to an input-enabled setting and language-inclusion based refinement. The original method does not assume determinism of processes in the system of constraints.

The preliminary version of this paper [LNW05] featured a stronger definition of mutual deadlock freeness: $E|S_1 \leqslant E_2$ and $E|S_2 \leqslant E_1$. Being stronger, this formulation also implies independent-implementability, but it rules out many useful

compositions as incompatible. The relativized version proposed here (B.2) is weaker, but still strong enough to imply independent implementability. As we have seen in the previous section, it behaves reasonably allowing roughly the same kind of compatible interfaces as interface automata. The present paper, completely rewritten, reworks the theory with this new characterization, adding associativity, refinement of interfaces, a new method for solving systems of inequalities, contravariant signature extension, and the correspondence to interface automata.

8 Conclusion

We have proposed an interface theory for distributed networks of asynchronous components modeled as I/O automata. The characteristic feature of our interfaces is an explicit separation of assumptions from guarantees. Apart from the usual engineering advantages offered by such a separation of concerns, it also allows modeling of families of interfaces implemented by software product lines.

We demonstrated that it is possible to build a reasonably behaved interface theory in an input-enabled setting, with language inclusion as refinement. We emphasize that our derivation of interface composition is systematic: we state requirements for composition and reduce the problem to finding a solution of a corresponding system of behavioral inequalities. We also discuss the generality of the constructed interface, concluding that it exhibits the weakest assumptions and the strongest guarantees that are possible with our requirements. Finally we describe a method for solving systems of inequalities arising in our setup and draw a formal correspondence between the present work and interface automata.

Paper C

Modal I/O Automata for Interface and Product Line Theories

Kim G. Larsen, Ulrik Nyman
Department of Computer Science,
Aalborg University, Denmark

Andrzej Wąsowski
Computational Logic and Algorithms Group,
IT University of Copenhagen, Denmark

Abstract

Alfaro and Henzinger use alternating simulation in a two player game as a refinement for interface automata [AH01]. We show that interface automata correspond to a subset of modal transition systems of Larsen and Thomsen [LT88], on which alternating simulation coincides with modal refinement. As a consequence a more expressive interface theory may be built, by a simple generalization from interface automata to modal automata. We define modal I/O automata, an extension of interface automata with modality. Our interface theory that follows can express liveness properties, disallowing trivial implementations of interfaces, a problem that exists for theories build around simulation preorders. In order to further exemplify the usefulness of modal I/O automata, we construct a behavioral variability theory for product line development.

Keywords: Modal I/O Automata, Interface Theory, Product Lines, Interface Automata, Modal Transition Systems

1 Introduction

An interface theory [AH01, CdAHS03, AHS02, LNW06b, ČVZ06, HRS05] is a type-system-like theory for component languages, where types (*interfaces*) describe components (*implementations*) with *composition* being the only operator available. A type error proves that either a component does not *conform* to its interface, or that two composed components are *incompatible*. Since the overall structure of these type systems is so simple, it is often accepted not to give typing rules explicitly when describing interface theories (for example [AH01, CdAHS03, AHS02, LNW06b, ČVZ06]), focusing instead on the essential ingredients of conformance, compatibility and composition.

Regular, non-component types are only applied to existing objects in program code. In contrast for interface theories it makes sense to discuss interfaces as specifications of application's architecture in isolation from actual source code. An interface abstracts the component in terms of the assumptions made by the component and the guarantees that it provides. One reasons about possible connections between component implementations (*compositions*) by using properties of composition of interfaces; most importantly *independent implementability* (that any implementations conforming to compatible interfaces are compatible) and generality properties (that the composition of interfaces produces an interface with the weakest assumptions and strongest guarantees).

We consider behavioral interface theories suitable for specification of communication protocols between components (web services or embedded systems). Such theories typically require a *contravariant* treatment of inputs and outputs to ensure deadlock-free implementations: inputs guaranteed by the specification are always offered by the implementation and that the implementation never produces more outputs than the specification. This observation led de Alfaro, Henzinger and colleagues [AH01, CdAHS03, AHS02] to a conclusion that game theoretical models of interaction are most suitable as building blocks for behavioral interface theories. While we do appreciate the values of the game theoretical formulations, we disagree with some claims in the above cited work and argue that game formulations are insufficient in themselves: there is a genuine value in combining the game theoretical approach with more traditional formulations based on transition systems, or more precisely on modal transition systems.

The two worlds of game models and modal transition systems convey largely orthogonal information about the moves of a system. Game models specify who has *control* over transitions, while modal transition systems focus on requirements, *modality*: which moves are allowed and which are required. In this paper we try to relate the two worlds, explain their weaknesses and their qualities. Eventually we combine them into a unified interface theory.

Game theoretical notions of conformance are often based on alternating simulation [AHKV98]. We show that alternating simulation in a two player setting, as used in interface automata [AH01, AH04], is just a special case of modal tran-

sition systems refinement developed by Larsen and Thomsen [LT88] in the late eighties. This suggests that the real value of the game theoretic approach to component theories does not lie in the use of alternating simulation, but in the use of *control* information in the composition synthesis algorithms.

Not surprisingly then, modal transition systems themselves cannot be used to build an interface theory, without adding control information. We build a new interface theory around *modal I/O automata,* which combine features of both game theoretic models and modal transition systems. Thanks to this new combination, our interfaces are now able to express liveness properties, which was impossible in existing interface theories (after this work has been completed we have learned about [CFN05], which achieves a similar effect in a different setting).

In order to further demonstrate the usefulness of our modal I/O automata, we construct a *product line* [Par76, CE00, PBvdL05] *theory.* In simple words a product line is a set of similar products built by combining *assets* from a common platform available in the development process. The differences between the products are referred to as *variability.* Our theory is a behavioral formalism for describing the variability of components. The theory supports deciding whether given requirements can be satisfied by choosing concrete instances from the set of available assets. This theory, though very small, is to the best of our knowledge one of the very few attempts at describing software product lines in a behavioral fashion, and unlike the previous work [LLW05], which takes a top-down approach to describing product families, it facilitates a bottom up construction of products, which is how product line development is more typically understood in the software engineering community. This contribution is not meant to be comprehensive, highly developed and well set in the tradition of the product line development. It should be understood as a simple example that emphasizes the semantic difference between modeling components in component based development and modeling assets for product family development. We do hope to extend this theory soon and report about it separately in detail.

The paper proceeds as follows. In the next section we shall explain the main results of the paper in nontechnical terms. Our main results concentrate in sections 3, 5 and 6. In Section 3 we draw a correspondence between the alternating simulation and observational modal refinement. In Section 4 modal I/O automata are defined, which are then used to construct an interface theory in Section 5 and a product line theory in Section 6. Sections 5 and 6 are largely independent, though they share a lot of intuitions. We conclude in Section 8.

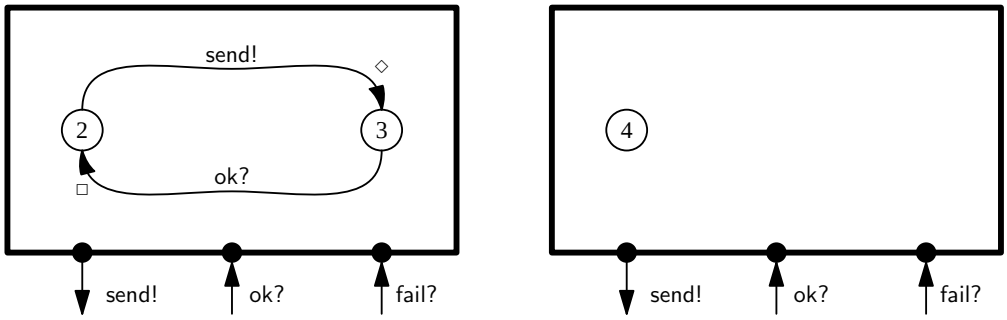


Figure C.1: The *Client* interface (left) and a trivial implementation of it (right).

2 Interface Automata vs Modal Automata: An Example

Consider an example interface automaton for a *Client* component (Fig. C.1 (left), originally presented in [AH01]). This simple model describes a component that occasionally may want to send a package, and once it has made the request it is ready to receive an acknowledgment. The signature of the interface also mentions a fail input, but the component is never able to receive it. This means that *Client* is only capable of interacting with network links that never fail.

In interface automata, due to a game theoretic semantics, all outputs are controlled by the component itself (called the *Output* player), while all inputs to such components are controlled by the environment player (called the *Input* player). An implementation conforms to the interface iff whenever some input is offered by the interface, then it is also offered by the implementation, and whenever an implementation produces any output, this output is also present in the interface (conformance formalized as alternating simulation [AHKV98]).

Such a notion of conformance implies that compatibility can be passed from interfaces to components: if there is no winning strategy for the input player that leads to a deadlock in the interface automaton, then there won't be such a strategy for the same player that interacts directly with any implementation. Similarly if there is no strategy for the output player that leads to an output that cannot be accepted by the environment, then there is also no such strategy for any of the implementations.

Unfortunately this notion of conformance, though very much safety oriented, does not enforce that the implementations take on any useful activities at all. Consider for example the diagram in the bottom of Fig. C.1. It presents a model of an implementation that does not perform any actions ever. In other words this is a network application that does not use the network at all. Still this new model conforms to its interface on the left, as in its initial state it does not add any illegal outputs and it offers all the inputs that were offered by the interface.

If we turn this into the terminology used in modal transition systems it means

that all the inputs are *required*, which is indicated by the □ (must) modality on the corresponding transition, and the outputs are *allowed*, which is indicated by the ◇ (may) modality on the transitions. In a modal transition systems perspective, conformance is based on modal refinement [LT88]. This refinement requires that whenever an implementation makes a step, then it must be possible to mimic it by an allowed transition of the specification; whenever the specification makes a required step it must be possible to match it with some required step of the corresponding state in the implementation. With the assignment of *may* to output transitions and *must* to input transitions this sounds nearly like the alternating simulation described above. In Section 3 we prove that indeed the two relations coincide if we require that the may transition relation is input-enabled.

Consequently modality gives strictly more modeling power than alternating refinement. Various modalities can be assigned to actions regardless of whom controls them. Instead of allowing all possible extensions on inputs, as in interface automata, the designer is able to control what extensions are allowed. For example we can change the *Client* model of Fig. C.1 to have a must modality (□) on the *send!* transition, which will have the effect that now all the implementations must be able to proceed producing an output. This would rule out trivial implementations as the one presented on the right side of Fig. C.1.

The game theoretic formulation of conformance gives a certain interpretation to inputs and outputs. Namely that inputs are *incoming requests* for service (for example remote procedure calls), while outputs are *outgoing requests* for service (also remote procedure calls, albeit in the other direction). With such an interpretation it becomes clear that removing services from the promised list should be illegal, while removing calls to external services is perfectly fine. This is exactly what alternating simulation achieves. What it misses is a more complex structure of communication.

In asynchronous systems some messages indeed convey calls for service, however many other return feedback from the services (return a value). When a given output models returning a value from a component, then clearly it should never be removed, as then the whole component becomes useless. Fig. C.2 illustrates another interface modeling a data link layer, which exploits the interplay between control and modality. The *must* modality is placed on transmt! transitions, as the data link layer would be useless if the implementation was permitted not to forward packets down the stack. Similarly the transition sending back the error message cannot legally be removed. At the same time the call for linkStatus! is a may transition as some implementations are allowed not to consult the hardware link explicitly to detect errors. Finally not all implementations are forced to be able to work with links that fail twice in a row, which is modeled by the second nack! transition being a may transition.

Now consider how the two interfaces of Fig. C.1 (left) and Fig. C.2 (top) should be composed. The composition resembles a product computation (taken separately for the may transition relation and the must transition relation). As

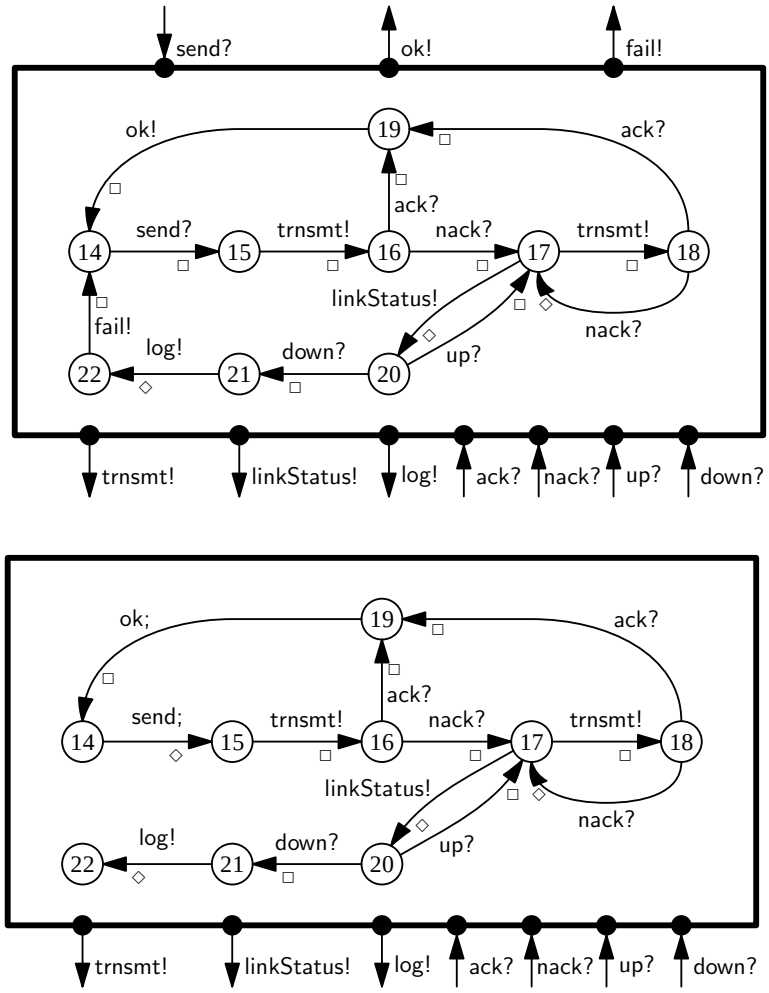


Figure C.2: *DataLink* layer with nontrivial modalities (top). Composition $DataLink \otimes Client$ (bottom). State 22 is an error state, where *DataLink* can produce the fail action, not accepted by *Client*.

a result we obtain the interface presented in the bottom of Fig. C.2. Because the client component was so weak, the ultimate interface shows a system that possibly may never do anything. However if *Client* will send some packets, these packets will certainly be processed by the composition, unless the hardware link is broken. In such a case it might be that the implementation will produce a fail! message which will cause a deadlock with the current version of the *Client* (this can happen when the composition is in state 22). Since we cannot modify the composed system we instead synthesize a new interface which restricts the use of the composition in order to guarantee error freeness. States of the composition that can experience deadlocks are called *error states*. We follow Alfaro and Henzinger in removing error states, and transitively all states from which error states can be reached by following *internally controllable transitions* of the

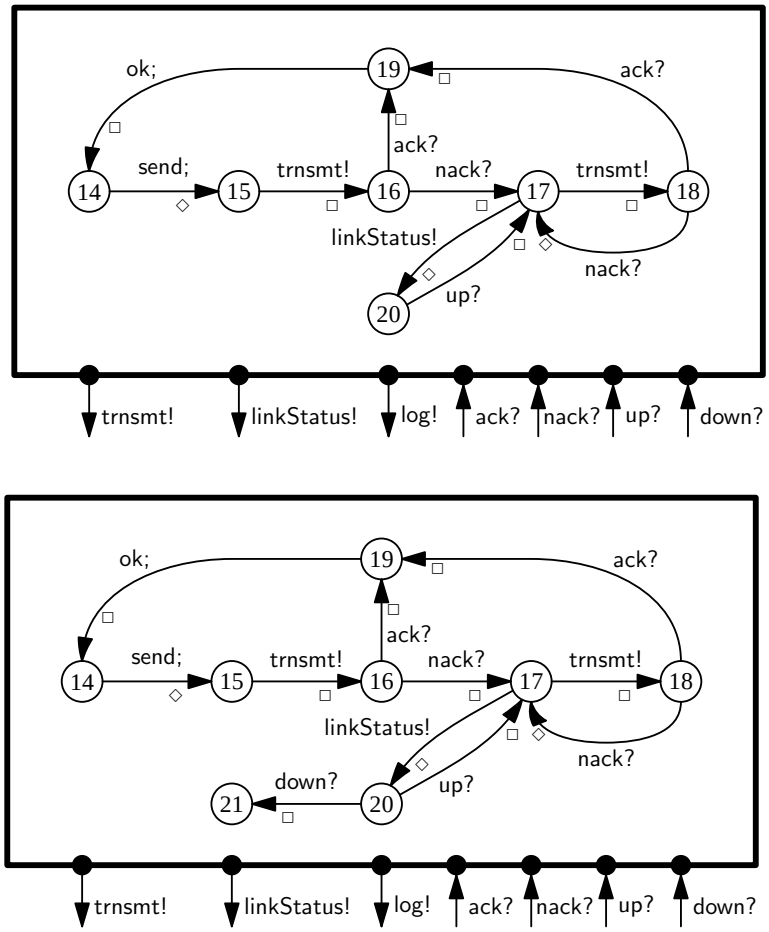


Figure C.3: Composed interfaces $LinkLayer \mid Client$ and variability models $LinkLayer \cdot Client$

component (outputs and internal actions). This leads to the interface on Fig. C.3 (top), expressing the fact that this component works well as long as the physical link never goes down.

The pruning mechanism described above would not be possible without the information describing which transitions are internally controllable being explicitly present in the model. It does not seem possible to compute the safe fragment of the product automaton, by just investigating the modalities of transitions. While we have said that modal refinement is strictly more expressive than alternating simulation, the control information of interface automata has its unique qualities too: it enables valuable synthesis algorithms not otherwise possible.

Let us now revisit the model of Fig. C.2 (top) giving it a different interpretation than previously. Instead of perceiving it as an abstraction of a component, we should now see it as a description of a set of components. A modal automaton

describes in fact a whole, often infinite, set of possible implementation automata[1]. One can think of them as all possible configurations of the model. This feature of modal automata suggests the possibility of using them as a behavioral formalism in describing variability in product lines.

A product line is a collection of products that are similar in that they offer overlapping functionality, and in that they are built from assets selected from a common platform. In here we want to describe both assets and the whole product line by modal I/O automata. If each of the assets is modeled as a modal I/O automaton we can model the capabilities of the family by composing these descriptions. However this time we would not be interested in a composition that guarantees compatible behavior of any selection of assets. It is normally expected that not all the assets in a product line platform are mutually compatible. Some of them will deadlock (for example a failing link layer and our *Client* component). The requirement for composing the variability descriptions is not to synthesize an interface that guarantees correctness of composition of all possible combination of assets, but to precisely describes what the correct combinations are: i.e. what are the deadlock free behaviors respecting the modalities that can be constructed with the available automata.

It turns out that a composition like that exists and it resembles the pruning of the product automaton for interface automata. The only difference is that now error states are the states where the error must be possible to realize (so one party must be required to produce an output that the other party must not be allowed to receive) and that we prune all the states from which reaching an error state is unavoidable (in our interface theory we have pruned states from which reaching errors might be possible).

The result of composing *Client* and *LinkLayer* using the variability model semantics is presented on the bottom of Figure C.3. This result contains a slightly bigger model than the interface automaton composition on the top. It states that there exists a pair of assets (implementations of *Client* and *LinkLayer*) such that it is able to accept a link down message without an error message. The transition with the down message was removed in the interface compositions as, for some pairs of implementations, it would lead to a deadlock.

Can a given specification be implemented by choosing components from available assets? Is the result of the composition the most general possible, containing all possible legal products? Can we find what the configuration of these elements should be? We address some of these questions in section 6, with an intention of elaborating more in upcoming work.

[1]This is also true for interface automata, though to a much lesser extent. Due to the lack of modality the set of implementations for an interface automaton is much simpler than it can be for a modal automaton.

3 Alternating Simulation vs Modal Refinement

Let us begin with defining modal automata, a version of modal transition systems [LT88] extended with signatures. A modal automaton has two transition relations indicating respectively allowed (*may*) and required (*must*) behavior.

Definition 1 (Modal Automaton) *A modal automaton S is a six tuple: $S = (states_S, start_S, ext_S, int_S, \longrightarrow_\diamond, \longrightarrow_\Box)$ where $states_S$ is a finite set of states, $start_S \in states_S$ is the initial state, ext_S and int_S are disjoint sets of external and internal actions and $act_S = ext_S \cup int_S$, $\longrightarrow_{\diamond S} \subseteq states_S \times act_S \times states_S$ is the may transition relation describing allowed behavior, and $\longrightarrow_{\Box S} \subseteq states_S \times act_S \times states_S$ is the must transition relation describing required behavior.*

Throughout the paper we sometimes use the symbols "!", "?" and ";" after an action. This is done in order to increase the readers intuition of whether the action is respectively an output, input or internal action. No symbol is used when the action can be of more than one type. These symbols could be left out completely as it is the identity of the action that is significant.

In the following we write $s \xrightarrow{\tau}{}^*_\Box s'$ meaning that there exists a sequence of internal *must* actions leading from s to s'. The same is defined for *may* transitions.

A modal automaton is *syntactically consistent* if everything that is required is also allowed, such that $\longrightarrow_\Box \subseteq \longrightarrow_\diamond$. In the following we only consider syntactically consistent modal automata. A modal automaton is an *implementation* if the two transition relations coincide.

A modal automaton describes a set of possible implementations. Simplistically when refining a modal automaton specification into an implementation one can remove a *may* transition, that does not have a corresponding *must* transitions or strengthen it into a *must* transition. In general this refinement is not syntactic, but behavioral, so it is not the syntactic transitions that are refined but the actual steps taken by the transition system. The same transition can be refined differently each time it is taken.

Definition 2 (Modal Refinement) *For a pair of modal automata S and T with the same signature, a binary relation $R \subseteq states_S \times states_T$ is a modal refinement if whenever sRt and $a \in act_S$ it holds that*

if $t \xrightarrow{a}_\Box t'$ then $\exists s'.s \xrightarrow{a}_\Box s'$ and $(s', t') \in R$.

if $s \xrightarrow{a}_\diamond s'$ then $\exists t'.t \xrightarrow{a}_\diamond t'$ and $(s', t') \in R$.

Modal refinement $\leq_{\mathrm{m}}$ is defined as the largest such relation. We say that a modal automaton S modally refines a modal automaton T, written $S \leq_{\mathrm{m}} T$, iff there exists a modal refinement containing $(start_S, start_T)$.

Observational modal refinement is a weaker refinement in which the two modal automata can take internal transitions, that cannot be directly observed by the other automaton. In absence of internal actions the observational refinement coincides with the non-observational one.

Definition 3 (Observational Modal Refinement) *For a pair of modal automata S and T with the same signature, a binary relation $R \subseteq states_S \times states_T$ is an observational modal refinement if whenever sRt and $a \in act_S$ it holds that*

if $t \xrightarrow{a}_{\Box} t'$ and $a \in ext_T$ then $\exists s'. s \xrightarrow{a}_{\Box} s' \wedge (s', t') \in R$.

if $s \xrightarrow{a}_{\Diamond} s'$ and $a \in ext_S$ then $\exists t'. t \xrightarrow{\tau}{}^*_{\Diamond} t'. \exists t''. t' \xrightarrow{a}_{\Diamond} t'' \wedge (s', t'') \in R$.

if $s \xrightarrow{a}_{\Diamond} s'$ and $a \in int_S$ then $\exists t'. t \xrightarrow{\tau}{}^*_{\Diamond} t'. (s', t') \in R$

Observational modal refinement $\leq^*_{\mathrm{m}}$ is defined as the largest such relation. We say that a modal automaton S observationally refines a modal automaton T if there exists an observational modal refinement containing $(start_S, start_T)$.

Interface Automata [AH01] can be considered a subset of modal automata in which the external actions ext_S are partitioned into inputs in_S and outputs out_S.

Definition 4 (Interface Automaton) *An* interface automaton *P is a tuple $P = (states_P, start_P, in_P, int_P, out_P, \rightarrow_P)$ where $states_P$ is a finite set of states, $start_P \in states_P$ is the initial state, in_P, out_P and int_P are three pairwise disjoint sets of input, output and hidden (internal) actions respectively, and $\rightarrow_P \subseteq states_P \times act_P \times states_P$ is the set of transitions where $act_P = in_P \cup out_P \cup int_P$.*

We require that the transition relation is input-deterministic such that for all $s, s', s'' \in states_P$ and all input actions $a \in in_P$ if $s \xrightarrow{a?} s'$ and $s \xrightarrow{a?} s''$ then $s' = s''$.

Similarly as for Modal Automata we define $s \xrightarrow{\tau}{}^* s'$ for Interface Automata to mean that there exists a sequence of internal transitions leading from s to s'. We define *alternating simulation* for interface automata as commonly used in software specification [AH04], which is slightly less general than the original [AH01]:

Definition 5 (Alternating Simulation) *For a pair of interface automata S and T with the same signature, a binary relation $R \subseteq states_S \times states_T$ is an alternating simulation if whenever sRt and $a \in act_S$ it holds that:*

if $t \xrightarrow{a?} t'$ and $a \in in_T$ then $\exists s'. s \xrightarrow{a?} s'$ and $(s', t') \in R$

if $s \xrightarrow{a!} s'$ and $a \in out_S$ then $\exists t'. t \xrightarrow{\tau}{}^* t'. \exists t''. t' \xrightarrow{a} t''$ and $(s, t'') \in R$

if $s \xrightarrow{a;} s'$ and $a \in int_S$ then $\exists t'. t \xrightarrow{\tau}{}^* t'$ and $(s', t') \in R$

Alternating simulation $\leq_{\mathrm{a}}$ is defined as the largest such relation. We say that S simulates T, written $S \leq_{\mathrm{a}} T$, if there exists an alternating simulation containing $(start_S, start_T)$.

In order to compare interface automata with modal automata, we construct a translation function $\mathcal{T}$ mapping from the former to the latter. The result of the translation always fulfills the conditions listed below. It is easy to see that for modal automata that fulfill these conditions a reversed mapping can be constructed, too.

1° The may transition relation is input enabled, meaning that for each state $s \in states_S$ and each input action $a \in in_S$ there exists a state s' and a may transition $s \xrightarrow{a?}_{\diamond} s'$

2° The constructed modal automaton is syntactically consistent: $\longrightarrow_{\Box} \subseteq \longrightarrow_{\diamond}$

3° Must transitions are only labeled by inputs: $\longrightarrow_{\Box S} \subseteq states_S \times in_S \times states_S$

Let s_{mayall} be a fresh state that allows all behavior but does not require any behavior. If U denotes the universe of all inputs, such that for all interface automata P, $in_P \in U$, then we define the translation function as follows:

$$\mathcal{T}(states_P, start_P, in_P, out_P, int_P, \rightarrow_P) = (states_S, start_S, ext_S, int_S, \longrightarrow_{\diamond}, \longrightarrow_{\Box})$$

where $states_S = states_P \cup \{s_{mayall}\}$, $start_S = start_P$, $ext_S = U \cup out_P$, $int_S = int_P$
and $s_1 \xrightarrow{a}{}^{S}_{\diamond} s_2$ if $s_1 \xrightarrow{a}{}^{P} s_2$ and $a \in out_P \cup int_P$
and $s_3 \xrightarrow{a}{}^{S}_{\Box} s_4$ and $s_3 \xrightarrow{a}{}^{S}_{\diamond} s_4$ if $s_3 \xrightarrow{a}{}^{P} s_4$ and $a \in in_P$
and $s_3 \xrightarrow{a}{}^{S}_{\diamond} s_{mayall}$ if $\forall s' \in states_P(s_3, a, s') \notin \rightarrow^P$ and $a \in U$,
and s_{mayall} is a fresh state such that $\forall a \in act_S . s_{mayall} \xrightarrow{a}{}^{S}_{\diamond} s_{mayall}$.

Theorem 1 *Alternating simulation and observational modal refinement coincide for interface automata in the following sense:*

$$\textit{for any two interface automata } S, T\textit{: } S \leq_{\mathrm{a}} T \textit{ iff } \mathcal{T}(S) \leq^*_{\mathrm{m}} \mathcal{T}(T) \qquad \text{(C.1)}$$

Theorem 1 suggests that the usefulness of game theoretical models for component theories does not lie in its conformance relation. The crux is the use of control information in synthesis algorithms, when paths to error states are pruned. If this is the case we can construct an interface theory based on modal refinement and modal automata augmented with control information. Since modal refinement is richer and we can use a generalization of the synthesis algorithm used for interface automata, we will obtain a more expressive interface theory.

The fact that alternating simulation coincides with the *observational* version of modal refinement is expected, because Definition 5 embeds a closure on internal transitions. In fact in the absence of internal actions alternating simulation coincides with the regular modal refinement, as described in Definition 1, which is easy to prove. In order to simplify the developments we use the regular modal refinement ($\leq_{\mathrm{m}}$) from now on, even though most of our theorems can reasonably be considered for the observational refinement ($\leq^*_{\mathrm{m}}$), too.

4 Modal I/O Automata

Let us now define modal I/O automata, an extension of modal automata with control information, that will be the main ingredients of our interface theory and the product line theory coming in the next sections.

Definition 6 *A modal I/O automaton S is a tuple $S = (states_S, start_S, in_S, out_S, int_S, \longrightarrow_\Diamond, \longrightarrow_\Box)$, where $states_S$ is a set of states, $start_S \in states_S$ is an initial state, in_S, out_S and int_S are pairwise disjoint sets of inputs, outputs and internal actions respectively ($act_S = in_S \cup out_S \cup int_S$), $\longrightarrow_{\Diamond S} \subseteq states_S \times act_S \times states_S$ is a may-transition relation, and $\longrightarrow_{\Box S} \subseteq states_S \times act_S \times states_S$ is a must-transition relation. Like previously we only consider syntactically consistent modal I/O automata here, so $\longrightarrow_\Box \subseteq \longrightarrow_\Diamond$.*

The composition for modal I/O automata combines both the modal aspects and the communications aspects. Two modal I/O automata S_1,S_2 are *composeable* iff their actions only overlap on complementary types: $(in_{S_1} \cup int_{S_1}) \cap (in_{S_2} \cup int_{S_2}) = \emptyset$ and $(out_{S_1} \cup int_{S_1}) \cap (out_{S_2} \cup int_{S_2}) = \emptyset$. The composition $S_1 \otimes S_2$ gives rise to a modal I/O automaton S such that $states_S = states_{S_1} \times states_{S_2}$, $start_S = (start_{S_1}, start_{S_2})$, $in_S = (in_{S_1} \setminus out_{S_2}) \cup (in_{S_2} \setminus out_{S_1})$, $out_S = (out_{S_1} \setminus in_{S_2}) \cup (out_{S_2} \setminus in_{S_1})$, $int_S = int_{S_1} \cup int_{S_2} \cup (in_{S_1} \cap out_{S_2}) \cup (out_{S_1} \cap in_{S_2})$. The transition relations are given by the following rules (see Fig. C.2 for an example):

$$\frac{s_1 \xrightarrow{a!}_\gamma s_1' \qquad s_2 \xrightarrow{a?}_\gamma s_2'}{s_1 \otimes s_2 \xrightarrow{a}_\gamma s_1' \otimes s_2'} \ \gamma \in \{\Box, \Diamond\} \qquad \frac{s_1 \xrightarrow{a?}_\gamma s_1' \qquad s_2 \xrightarrow{a!}_\gamma s_2'}{s_1 \otimes s_2 \xrightarrow{a}_\gamma s_1' \otimes s_2'} \ \gamma \in \{\Box, \Diamond\}$$

$$\frac{s_1 \xrightarrow{a}_\gamma s_1' \qquad a \notin act_{S_2}}{s_1 \otimes s_2 \xrightarrow{a}_\gamma s_1' \otimes s_2} \ \gamma \in \{\Box, \Diamond\} \qquad \frac{s_2 \xrightarrow{a}_\gamma s_2' \qquad a \notin act_{S_1}}{s_1 \otimes s_2 \xrightarrow{a}_\gamma s_1 \otimes s_2'} \ \gamma \in \{\Box, \Diamond\}$$

For technical reasons (efficiency and simplicity) we always assume that unreachable states are removed after computing a composition (both here and in later sections). The following theorem is a simple corollary from the general fact that the modal refinement is a precongruence [BL90, Lar89]:

Theorem 2 *Modal refinement is a precongruence with respect to the above composition operator: for any four modal I/O automata T_1, T_2, S_1, S_2 such that $T_1 \leq_\mathrm{m} S_1$ and $T_2 \leq_\mathrm{m} S_2$ it holds that $T_1 \otimes T_2 \leq_\mathrm{m} S_1 \otimes S_2$.*

The composition operator ($\otimes$) defined above corresponds to a usual composition of software (hardware) *components*. Whenever we use it below we mean an unrestricted connection of components, which does not preclude deadlocks or other kinds of errors. We shall soon introduce two seemingly similar composition operators, ($|$) and ($\cdot$) having a very different use. In fact they are algorithms synthesizing *specifications* of how a result of simple composition ($\otimes$) should be used in order to guarantee the absence of certain errors.

5 A Modal Interface Theory

Interface theories support component based development. The aim is to specify component interfaces and from these interfaces to derive the interfaces of composite components. The novel aspect of the interface theory presented here is that

the components can specify both required and allowed behavior, consequently it is suitable for expressing liveness properties.

In our specific interface theory an interface is given by a modal I/O automaton. A given interface specifies a set of potential implementations (concrete implementations have identical transition relations $\longrightarrow_\diamond = \longrightarrow_\Box$). The goal of our interface theory is to be able to use interface descriptions to describe legal implementations of components in a component based system. The implementation relation, the relation that specifies which implementations conform to a given interface description is modal refinement $\leq_{\mathrm{m}}$. From the interface descriptions of two components it should be possible to derive the interface of the combined component. This is done without knowing more about the implementations, than the fact that they conform to their individual interface specification.

The result of composing two interfaces is a subset of the result of composing two modal I/O automata, in which all possible internally controllable paths leading to error states are removed. An *error state* is a state in which one component can output something that the other component might be unable to receive:

$$\begin{aligned} err^{\mathrm{i}}_{S_1,S_2} = \{(s_1,s_2) \in states_{S_1 \otimes S_2} \mid \text{there exists } a \in int_{S_1 \otimes S_2} \text{ and states } s_1',\ s_2' \\ \text{such that } (s_1 \xrightarrow{a!}{}^{S_1}_{\diamond}\ s_1' \text{ and } s_2 \not\xrightarrow{a?}{}^{S_2}_{\Box}) \text{ or } (s_2 \xrightarrow{a!}{}^{S_2}_{\diamond}\ s_2' \text{ and } s_1 \not\xrightarrow{a?}{}^{S_1}_{\Box})\} \end{aligned} \quad \text{(C.2)}$$

State 22 on Fig. C.2 is an error state, witnessed by the fail action.

We are now ready to define the set of states of the composition:

$$states_{S_1|S_2} = \bigcap_{n=0}^{\infty} prune_{\mathrm{i}}^n(states_{S_1 \otimes S_2} \setminus err^{\mathrm{i}}_{S_1,S_2}) \ , \quad \text{(C.3)}$$

where $prune_{\mathrm{i}}(S) = \{s \in S \mid \forall s' \forall a \in int_{S_1 \otimes S_2}.\ s \xrightarrow{a}_\diamond s' \text{ implies } s' \in S\}$, which is a monotonic function that removes, from the set of states S, all those states that in one internally controllable step may reach a state that is not in S.

See Figure C.3 (left) for an example of how pruning works. State 22 has been removed as an error state, then state 21 was pruned as an error state can be reached from it by the internally controllable transition log!. Then all transitions involving states 21 and 22 were removed. State 20 remains in the result as the must transition labeled down is externally controllable.

Definition 7 (Composition) *The composition of two interfaces S_1 and S_2 is defined if S_1 and S_2 are composable modal I/O automata and $start_{S_1 \otimes S_2} \in states_{S_1|S_2}$ (see above). The composition results in a modal I/O automaton $S_1|S_2$ such that* $S_1|S_2 = (states_{S_1|S_2}, start_{S_1 \otimes S_2}, in_{S_1 \otimes S_2}, out_{S_1 \otimes S_2}, int_{S_1 \otimes S_2}, \longrightarrow_\diamond^{S_1 \otimes S_2} \cap (states_{S_1|S_2} \times act_{S_1 \otimes S_2} \times states_{S_1|S_2}), \longrightarrow_\Box^{S_1 \otimes S_2} \cap (states_{S_1|S_2} \times act_{S_1 \otimes S_2} \times states_{S_1|S_2}))$.

Two interfaces are compatible if the set of states resulting from composition, $states_{S_1|S_2}$, contains the initial state $(start_{S_1}, start_{S_2})$.

A desirable property of an interface theory is that components can be implemented independently of each other once the specifications are known. The following theorem formally states that this theory satisfies the property.

Theorem 3 (Independent Implementability) *For any two compatible interfaces S_1, S_2 and for any two implementations I_1, I_2, $I_1 \leq_{\mathrm{m}} S_1$ and $I_2 \leq_{\mathrm{m}} S_2$, it holds that $I_1 \otimes I_2 \leq_{\mathrm{m}} S_1|S_2$.*

This has three implications. First, $I_1 \otimes I_2$ would deliver all the required behavior promised by $S_1|S_2$ as long as it interacts with an environment obeying $S_1|S_2$. Second, $I_1 \otimes I_2$ will not do anything that $S_1|S_2$ would not allow in such an environment. Third, since $S_1|S_2$ does not contain error states then $I_1 \otimes I_2$ will not deadlock.

Theorem 4 (Deadlock Freeness Preservation) *For any two compatible interfaces S_1, S_2, any two implementations I_1, I_2, so $I_1 \leq_{\mathrm{m}} S_1$ and $I_2 \leq_{\mathrm{m}} S_2$, and any interface T compatible with $S_1|S_2$, if $T \otimes (S_1|S_2)$ has no reachable error states then $T \otimes (I_1 \otimes I_2)$ has no reachable error states.*

Finally the composition operator (|) is commutative and associative up to graph isomorphism.

6 A Product Line Theory

In product line development one typically maintains a family of existing *assets* that are composed in a bottom-up fashion in order to build a product. Here we assume that existing assets are sufficient to build the product and no genuinely new programming is required. Assets are organized in small subfamilies, that can be thought of as configurable components. Choosing an asset from a subfamily is a configuration process. We model subfamilies as modal I/O automata, and call them *variability models*, to distinguish them from interfaces. The configuration process amounts to finding a suitable modal refinement of a variability model.

There is a need for a mechanism for composing variability models, to enable reasoning about the products that can be constructed using available assets. As in the interface theory we are interested in computing the legal uses for the composition of two models, without reaching error states. However we weaken the requirement this time: we do not require that *all* possible pairs of implementations give an error free composition, but only that there *exists* a pair of implementations that can avoid errors under a suitable use.

Two variability models are composable if their input, output and hidden actions do not overlap (the general rule for modal I/O automata). Two composable families can be composed, resulting in a description of a higher level component family. The signature of this variability model is found in the same way as for modal I/O automata. The requirement for the description of this more abstract

family is that a specification that refines its description can be realized by choosing some concrete implementations from both lower level families involved. So that in effect one can configure the final product by configuring the abstract composed variability model, being sure that the selected configuration can be refined to configurations of each of the smaller components, available in the collection of assets. We give a sufficient condition for a refinement of a variability model to be decomposable.

The ultimate composition closely resembles the composition (|) for interface automata: it uses the regular modal I/O automata composition ($\otimes$) first and then removes error states. However now only internally controllable *required transitions* are pruned, while in the interface theory we had also removed states reachable by *allowed executions* of the same kind. The very existence of allowed internally controlled execution to an error state was considered dangerous in the interface theory—it is not in the product line theory. This is because we are not interested in eliminating errors by all means, but only in making sure that there exist error-free realizations of the specification. For two syntactically composable variability models we define the set of error states, $err^{\mathrm{v}}_{S_1,S_2}$, to be:

$$err^{\mathrm{v}}_{S_1,S_2} = \{(s_1,s_2) \in states_{S_1\otimes S_2} \mid \text{ there exists } a \in int_{S_1\otimes S_2} \text{ and states } s_1', s_2' \text{ such that } (s_1 \xrightarrow{a!}_{\Box} s_1' \text{ and } s_2 \not\xrightarrow{a?}_{\Diamond}) \text{ or } (s_1 \not\xrightarrow{a?}_{\Diamond} \text{ and } s_2 \xrightarrow{a!}_{\Box} s_2')\} \quad \text{(C.4)}$$

In Figure C.2 (right) state 22 is still an error state, though for a different reason than previously: in state 22 the *LinkLayer must* be able to produce fail, but the *Client* is *not allowed* to receive it. If a product of two variability models contains an error state it means that there exist configurations of composed assets that cannot safely work together. However, in the same spirit as in the interface theory, we can compute the set of legal uses that guarantee that there *exist* pairs of compatible configurations to interact with them. We remove from the product $S_1 \otimes S_2$ all the states that according to the variability specification *must* be able to reach an error state. If there is no states left then the two variability models are *incompatible*. Otherwise we arrive at a specification of states and transitions among the compatible states that constraint possible legal implementations obtained from these two families. Formally:

$$states_{S_1\cdot S_2} = \bigcap_{n=0}^{\infty} prune^{n}_{\mathrm{v}}(states_{S_1\otimes S_2} \setminus err^{\mathrm{v}}_{S_1,S_2}) \ , \quad \text{(C.5)}$$

where $prune_{\mathrm{v}}(S) = \{s \in S \mid \forall s'.\, \forall a \in int_{S_1\otimes S_2} \cup out_{S_1\otimes S_2}.\, s \xrightarrow{a}_{\Box} s' \text{ and } s' \in S\}$. We compute the two transition relations for the composition, by projecting the transition relations of the parallel composition $S_1 \otimes S_2$ onto the new set of states:

$$\longrightarrow^{S_1\cdot S_2}_{\Diamond} = \longrightarrow^{S_1\otimes S_2}_{\Diamond} \cap (states_{S_1\cdot S_2} \times act_{S_1\otimes S_2} \times states_{S_1\cdot S_2}) \quad \text{(C.6)}$$

$$\longrightarrow^{S_1\cdot S_2}_{\Box} = \longrightarrow^{S_1\otimes S_2}_{\Box} \cap (states_{S_1\cdot S_2} \times act_{S_1\otimes S_2} \times states_{S_1\cdot S_2}) \ . \quad \text{(C.7)}$$

Finally we can state the complete result of the composition: a modal I/O automaton $S_1 \cdot S_2$ such that $S_1 \cdot S_2 = (states_{S_1 \cdot S_2}, (start_{S_1}, start_{S_2}), in_{S_1 \otimes S_2}, out_{S_1 \otimes S_2}, int_{S_1 \otimes S_2}, \longrightarrow_{\diamond}^{S_1 \cdot S_2}, \longrightarrow_{\Box}^{S_1 \cdot S_2})$ and all the components are defined above.

Definition 8 *Two variability models are compatible if they are composable and their composition is nonempty.*

It turns out that *observationally consistent* refinements of compositions of variability models are realizable with existing assets. We define observational consistency for states of a single automaton. Let $t \xrightarrow{A}_{\Box}{}^{*} t'$ mean that t' is reachable from t via a possible empty sequence of required transitions labeled by possibly different actions from a set A.

Definition 9 *Let T be a modal automaton and let $A \subseteq act_T$ be a set of actions. A relation $C \subseteq states_T \times states_T$ is an observational consistency relation with respect to A if for any pair of states $(t_1, t_2) \in C$ the following two properties hold:*

1° $\forall t_1'.$ *if* $t_1 \xrightarrow{A}_{\Box}{}^{*} t_1'$ *then* $\forall a \notin A. \forall t_1''. t_1' \xrightarrow{a}_{\Box} t_1''$ *implies* $\exists t_2'. t_2 \xrightarrow{a}_{\diamond} t_2' \wedge (t_1'', t_2') \in C$.

2° $\forall t_2'.$ *if* $t_2 \xrightarrow{A}_{\Box}{}^{*} t_2'$ *then* $\forall a \notin A. \forall t_2''. t_2' \xrightarrow{a}_{\Box} t_2''$ *implies* $\exists t_1'. t_1 \xrightarrow{a}_{\diamond} t_1' \wedge (t_1', t_2'') \in C$.

Two states are observationally consistent if there exists an observational consistency relation relating them. A set of states is said to be observationally consistent with respect to A if all possible pairs of states from the set are observationally consistent with respect to A. An automaton T is observationally consistent with respect to A iff the set $\{start_T\}$ is an observationally consistent set.

The following theorem states the existence of decomposition formally:

Theorem 5 (Decomposability) *Let T_1, T_2 be deterministic composable variability models, and S be a configuration (a deterministic variability model itself) such that $S \leq_{\mathrm{m}} T_1 \cdot T_2$, and T_1, S are observationally consistent with respect to $act_{T_1} \setminus act_{T_2}$ and T_2, S are observationally consistent with respect to $act_{T_2} \setminus act_{T_1}$. Then there exist S_1 and S_2 such that $S_1 \leq_{\mathrm{m}} T_1$ and $S_2 \leq_{\mathrm{m}} T_2$ and $S_1 \otimes S_2 \leq_{\mathrm{m}} S$.*

A version of the theorem, not requiring observational consistency, does not hold, which can be demonstrated with a counter-example, not included here.

An important corollary is that the decomposition can be carried over down to precise configurations: if a concrete configuration of a product is required, then there exist concrete configurations of assets to realize it. The question whether a specification is realizable with given assets is reduced to establishing observational consistency and a modal refinement between the postulated requirement

and the variability model. Consequently the abstract variability model can be communicated to configuration engineers and used to configure final products.

Let us close our discussion with a statement that the $(\cdot)$ operator is general enough to describe all implementations safely realizable with existing assets.

Theorem 6 (Completeness) *For any two compatible variability models T_1, T_2 and any two compatible concrete implementation specifications I_1, I_2, where $I_1 \leq_{\mathrm{m}} T_1$ and $I_2 \leq_{\mathrm{m}} T_2$ it holds that $I_1 \cdot I_2 \leq_{\mathrm{m}} T_1 \cdot T_2$.*

7 Conclusion & Future Work

We have investigated the relation between alternating simulation as used in interface automata and observational modal refinement, concluding that former is a case of the latter. We have argued that the strength of the game theoretic approach to interface theories does not lie in alternating refinement itself, but in the labeling of transitions with control information; in partitioning the actions into internally and externally controllable. We have extended modal transition systems with this information and demonstrated that in this way interface theories tracking liveness properties, can be built. Finally we have presented a product line theory describing variability in behavior of component families.

In the future we would like to extend the product line theory of Section 6 to a full featured theory based on observational modal refinement and study its properties in depth. Also it appears interesting to investigate the relation between the general notion of alternating refinement [AHKV98] and (modal) transition systems, lifting the restrictions accepted in Section 3 after the interface automata model.

8 Proofs

This appendix contains proofs of theorems and lemmas, along with some counterexamples for negative claims or one-way implications. *The appendix is not an integral part of the paper, and reading it is not required in order to assess the value of the results.*

8.1 Appendix for Section 3

This section uses formulations of Alternating Simulation and Observational Modal Refinement with ϵ-*closure*(s) instead of $s \xrightarrow{\tau}{}^{*}$.

Proof 1 (of Theorem 1) *The proof will be divided into two directions. First we will prove that*

$$\forall S, T \in IA.\ S \leq_{\mathrm{a}} T \implies \mathcal{T}(S) \leq_{\mathrm{m}}^{*} \mathcal{T}(T).$$

We will prove this by showing that alternating simulation is a subset of observational modal refinement on the translation of IA: $\leq_{\mathrm{a}} \subseteq \leq_{\mathrm{m}}^{*}$. *This will be shown by showing that the following relation is a modal refinement.*

$$R = \{(s,t) | \exists \hat{s}, \hat{t}. s = \mathcal{T}(\hat{s}) \wedge t = \mathcal{T}(\hat{t}) \wedge \hat{s} \leq_{\mathrm{a}} \hat{t}\} \cup \{(s, s_{mayall}) | s \in states_S\}$$

This is shown in three different cases, one for each of the rules that define observational modal refinement.

1° **Must transition, external action:** *Take* $t.t \xrightarrow{a}_{\Box} t' \wedge a \in ext_T$. *We can conclude from the definition of translation that this case only exists for* $a \in in_T$. *By* R *we have that* $\exists \hat{t}.\hat{t} \xrightarrow{a?} \hat{t}'$. *From the definition of Alternating Simulation we have that* $\exists \hat{s}.\hat{s} \xrightarrow{a?} \hat{s}' \wedge (s', t') \in R$. *By translation we have that* $s \xrightarrow{a}_{\Box} s'$ *and this implies that* $(s', t') \in R$.

2° **May transition, external action:** *Take* $s.s \xrightarrow{a}_{\Diamond} s' \wedge a \in out_S \cup in_S$ *it means, by* R, *that* $\exists \hat{s}.\hat{s} \xrightarrow{a} \hat{s}'$

- *2.1* $a \in out_S \wedge \hat{s} \xrightarrow{a!} \hat{s}'$, *by* $\hat{s} \leq_{\mathrm{a}} \hat{t}$ *and the definition of alternating simulation we have that* $\hat{t} \xrightarrow{a!} \hat{t}' \wedge \hat{s}' \leq_{\mathrm{a}} \hat{t}'$. *By translation we have* $t \xrightarrow{a}_{\Diamond} t'$ *this all implies that* $(s', t') \in R$.
- *2.2* $a \in in_S \wedge \hat{s} \xrightarrow{a?} \hat{s}' \wedge \hat{t} \xrightarrow{a?} \hat{t}'$, *by* $\hat{s} \leq_{\mathrm{a}} \hat{t}$, *the definition of alternating simulation and the fact that IA are input deterministic we have that* $\hat{s} \xrightarrow{a?} \hat{s}' \wedge \hat{s}' \leq_{\mathrm{a}} \hat{t}'$ *and this implies that* $(s', t') \in R$.
- *2.3* $a \in in_S \wedge \hat{s} \xrightarrow{a?} \hat{s}' \wedge \hat{t} \not\xrightarrow{a?}$, *by translation we have* $t \xrightarrow{a?}_{\Diamond} s_{mayall}$ *and by definition of* R *we have that* $(s, s_{mayall}) \in R$

3° **May transition, internal action:** *Take* $s.s \xrightarrow{a}_{\diamond} s' \wedge a \in int_S$ *it means, by* R *and translation, that* $\hat{s} \xrightarrow{a;} \hat{s}' \wedge s = \mathcal{T}(\hat{s})$*. By the definition of alternating simulation we have that* $\exists \hat{t}'.\hat{t} \xrightarrow{\tau}{}^* \hat{t}' \wedge \hat{s}' \leq_{\mathrm{a}} \hat{t}'$*. By translation we have that* $\exists t'.t \xrightarrow{\tau}{}^*_{\diamond} t'.t' = \mathcal{T}(\hat{t}')$*. This all implies* $(s', t') \in R$*.*

We will now prove the other direction:

$$\forall S, T \in IA.\ S \leq_{\mathrm{a}} T \iff \mathcal{T}(S) \leq^*_{\mathrm{m}}{}^{\mathcal{T}(T)}.$$

We will prove this by showing that observational modal refinement, on the translation of IA, is a subset of alternating : $\leq^*_{\mathrm{m}} \subseteq \leq_{\mathrm{a}}$*. This will be shown by showing that the following relation is an alternating simulation.*

$$Q = \{(\hat{s}, \hat{t}) | \exists s, t.s = \mathcal{T}(\hat{s}) \wedge t = \mathcal{T}(\hat{t}) \wedge s \leq^*_{\mathrm{m}}{}^{t}\}$$

This will be split into three cases, one for each of the rules in the definition of Alternating Simulation.

1° *Take* $\hat{t} \xrightarrow{a?} \hat{t}'$ *by* Q *and translation we have that* $a \in in_T \wedge t \xrightarrow{a}_{\Box} t'$*. We have by* $s \leq^*_{\mathrm{m}}{}^{t}$ *and the definition of Observational Modal Refinement that* $\exists s'.s \xrightarrow{a}_{\Box} s' \wedge s' \leq_{\mathrm{m}} t'$ *and by translation we have that* $\hat{s} \xrightarrow{a} \hat{s}'$ *where* $s' = \mathcal{T}(\hat{s}')$ *which implies that* $(\hat{s}', \hat{t}') \in Q$*.*

2° *Take* $\hat{s} \xrightarrow{a!} \hat{s}'$*, by* Q *and translation we have that* $a \in out_S.s \xrightarrow{a}_{\diamond} s'$*. We have by* $s \leq^*_{\mathrm{m}}{}^{t}$ *and the definition of Observational Modal Refinement that* $\exists t'.t \xrightarrow{\tau}{}^*_{\diamond} t'$*.* $\exists t''.t' \xrightarrow{a}_{\diamond} t''$ *and* $s' \leq^*_{\mathrm{m}}{}^{t''}$*. By translation we have that this will give rise to a sequence of internal transitions followed by an a transition such that we know that* $\exists \hat{t}'.\hat{t} \xrightarrow{\tau}{}^* \hat{t}'.\hat{t}' \xrightarrow{a!} \hat{t}'' \wedge \hat{s}' \leq_{\mathrm{a}} \hat{t}''$ *This all implies that* $(\hat{s}', \hat{t}'') \in Q$*.*

3° *Take* $\hat{s} \xrightarrow{a;} \hat{s}'$*. By* Q *and by translation we have that* $a \in int_S \wedge s \xrightarrow{a}_{\diamond} s'$ *We have by* $s \leq^*_{\mathrm{m}}{}^{t}$ *and the definition of Observational Modal Refinement that* $\exists t'.t \xrightarrow{\tau}{}^*_{\diamond} t' \wedge s' \leq^*_{\mathrm{m}}{}^{t'}$*. By translation we know that this sequence of zero or more internal transitions will give rise to an identical sequence of internal transitions such that* $\exists \hat{t}'.\hat{t} \xrightarrow{\tau}{}^* \hat{t}'$ *and* $\hat{s}' \leq_{\mathrm{a}} \hat{t}'$*. This all implies that* $(\hat{s}', \hat{t}') \in Q$

□

8.2 Appendix for Section 4

Lemma 1 *For any two composeable and syntactically consistent modal I/O automata* S_1*,* S_2 *their parallel composition* $S_1 \otimes S_2$ *is also syntactically consistent.*

8.3 Appendix for Section 5

Proof 2 (of Theorem 3) *This theorem is proven by showing that the relation R is a modal refinement:*

$$R = \{(i,s) \in states_{I_1 \otimes I_2} \times states_{S_1|S_2} | i = (i_1,i_2) \wedge s = (s_1,s_2) \wedge i_1 \leq_{\mathrm{m}} s_1 \wedge i_2 \leq_{\mathrm{m}} s_2\}$$

The proof is divided into two cases, one for each of the rules in the definition of modal refinement.

1° $s \xrightarrow{a}_{\Box} s'$. *This means that* $(s_1,s_2) \xrightarrow{a}_{\Box} (s_1',s_2')$.

We want to show that $\exists i'.i \xrightarrow{a}_{\Box} i' \wedge (i',s') \in R$. *This will be divided into five sub cases depending on how* $(s_1,s_2) \xrightarrow{a}_{\Box} (s_1',s_2')$ *is achieved. Several of these cases are symmetric versions of each other.*

1.1 $s_1 \xrightarrow{a!}_{\Box} s_1' \wedge a \in int_{S_1|S_2}$. *We know that* $s_2 \xrightarrow{a?}_{\Box} s_2'$ *must exists, else the output transition would have been pruned. We know* $i_1 \leq_{\mathrm{m}} s_1 \wedge i_2 \leq_{\mathrm{m}} s_2$ *which gives us* $i_1 \xrightarrow{a!}_{\Box} i_1' \wedge i_2 \xrightarrow{a?}_{\Box} i_2'$. *So take* $i = (i_1',i_2')$, *by definition of* $I_1 \otimes I_2$ *we have that* $i \xrightarrow{a}_{\Box} i'$ *and this implies that* $(i',s') \in R$.

1.2 This case is completely symmetric, where it is s_2 that outputs.

1.3 $s_1 \xrightarrow{a!}_{\Box} s_1' \wedge a \in out_S \wedge a \in ext_{S_1|S_2}$ *by* $i_1 \leq_{\mathrm{m}} s_1$ *we have that* $i_1 \xrightarrow{a!}_{\Box} i_1' \wedge i_1' \leq_{\mathrm{m}} s_1'$. *Also in this case we have, by composability, that* $s_2' = s_2 \wedge i_2' = i_2$ *and* $(i_1,i_2) \xrightarrow{a!}_{\Box} (i_1',i_2)$. *For* $i' = (i_1',i_2) \wedge s' = (s_1',s_2)$ *this all implies that* $(i',s') \in R$.

1.4 $s_1 \xrightarrow{a?}_{\Box} s_1' \wedge a \in in_S \wedge a \in ext_{S_1|S_2}$. *This case is symmetric with the previous case.*

1.5 $s_1 \xrightarrow{a;}_{\Box} s_1' \wedge a \in int_S \wedge a \in int_{S_1|S_2}$. *This case is symmetric with the previous case. All three cases also have symmetric cases where the transition in question is part of* S_2.

2° $i \xrightarrow{a}_{\Diamond} i'$ *this means that* $(i_1,i_2) \xrightarrow{a}_{\Diamond} (i_1',i_2')$.

We want to show that $\exists s'.s \xrightarrow{a}_{\Diamond} s' \wedge (i',s') \in R$. *This will be divided into five sub cases depending on how* $(i_1,i_2) \xrightarrow{a}_{\Diamond} (i_1',i_2')$ *is achieved. Several of these cases are symmetric versions of each other.*

2.1 $i_1 \xrightarrow{a!}_{\Diamond} i_1' \wedge i_2 \xrightarrow{a?}_{\Diamond} i_2'$. *By R and the definition of* $\leq_{\mathrm{m}}$*we have that* $s_1 \xrightarrow{a!}_{\Diamond} s_1' \wedge s_2 \xrightarrow{a!}_{\Diamond} s_2' \wedge i_1' \leq_{\mathrm{m}} s_1' \wedge i_2' \leq_{\mathrm{m}} s_2'$ *which gives us that* $((i_1',i_2'),(s_1',s_2')) \in R$.

2.2 This case is completely symmetric, where it is i_2 that outputs.

2.3 $i_1 \xrightarrow{a!}_{\Diamond} i_1' \wedge a \in out_I \wedge a \in ext_{I_1 \otimes I_2}$ *by* $i_1 \leq_{\mathrm{m}} s_1$ *we have that* $s_1 \xrightarrow{a!}_{\Diamond} s_1' \wedge i_1' \leq_{\mathrm{m}} s_1'$. *Also in this case we have, by composability, that* $s_2' = s_2 \wedge i_2' = i_2$ *and* $(s_1,s_2) \xrightarrow{a!}_{\Diamond} (s_1',s_2)$. *For* $i' = (i_1',i_2) \wedge s' = (s_1',s_2)$ *this all implies that* $(i',s') \in R$.

2.4 $i_1 \xrightarrow{a?}_{\diamond} i_1' \wedge a \in in_I \wedge a \in ext_{I_1 \otimes I_2}$. *This case is symmetric with the previous case.*

2.5 $i_1 \xrightarrow{a;}_{\diamond} i_1' \wedge a \in int_I \wedge a \in int_{I_1 \otimes I_2}$. *This case is symmetric with the previous case. All three cases also have symmetric cases where the transition in question is part of* I_2.

□

Proof 3 (of Theorem 4) *The proof proceeds as a contrapositive proof in which we show that if an error state was reachable in* $T \otimes (I_1 \otimes I_2)$ *then an error state would also be reachable in* $T \otimes (S_1|S_2)$. *There are two ways in which an error state could be reachable in* $T \otimes (I_1 \otimes I_2)$.

1° $err^{\mathrm{i}}_{T,(I_1 \otimes I_2)} \cap reachable(T \otimes (I_1 \otimes I_2))$*is non empty.*

2° $\Pi_2(reachable(T \otimes (I_1 \otimes I_2))) \cap err^{\mathrm{i}}_{I_1,I_2}$ *is non empty.*

Contrapositive proof:

1° *Assume that* $(t,i) \in err^{\mathrm{i}}_{T,(I_1 \otimes I_2)}$ *and that* (t,i) *is reachable. No we want to show that* $\exists (t,s) \in err^{\mathrm{i}}_{T,(S_1|S_2)}$ *and that* (t,s) *is reachable.*

Because t *is reachable and* $I_1 \otimes I_2 \leq_{\mathrm{m}} S_1|S_2$ *(Theorem 3) we know that* $\exists s \in states_{S_1|S_2}$ *and* $i \leq_{\mathrm{m}} s \wedge s$ *is reachable by may transitions in* $S_1|S_2$.

1.1 $t \xrightarrow{a!}_{\diamond} t' \wedge i \not\xrightarrow{a?}_{\square} \wedge a \in int_{T \otimes (I_1 \otimes I_2)}$ *but then* $s \not\xrightarrow{a?}_{\square}$. *We now need to argue that* (t,s) *is reachable by may transitions. This follows from* $I_1 \otimes I_2 \leq_{\mathrm{m}} S_1|S_2$ *(Theorem 3). Because of consistency we only consider may transitions.*

Executions of T *and* $I_1 \otimes I_2$ *is a sequence of may transitions of* T *and* $I_1 \otimes I_2$. *All the may transitions of* $I_1 \otimes I_2$ *can be matched by may transitions of* $S_1|S_2$

1.2 $i \xrightarrow{a!}_{\diamond} i' \wedge t \not\xrightarrow{a?}_{\square} \wedge a \in int_{T \otimes (I_1 \otimes I_2)}$. *The argument is identical to the previous case.*

2° *Assume that* $i_1 \xrightarrow{a!}_{\diamond} i_1' \wedge i_2 \not\xrightarrow{a?}_{\square}$ *and* $\exists t.(t, i_1, i_2)$ *is reachable. This implies that* $s_1 \xrightarrow{a!}_{\diamond} s_1' \wedge s_2 \not\xrightarrow{a?}_{\square}$. *So we can conclude that an error state would be reachable in* $T \otimes (S_1|S_2)$ *in this case.*

Lemma 2 *For any two composeable and syntactically consistent modal interface automata* S_1, S_2 *their parallel composition* $S_1|S_2$ *is also syntactically consistent.*

Theorem 7 (Associativity) $\forall S_1, S_2, S_3$. *pairwise compatible* $S_1|(S_2|S_3)$ *is isomorphic with* $(S_1|S_2)|S_3$.

8.4 Appendix for Section 6

Lemma 3 *For any two composeable and syntactically consistent modal variability models S_1, S_2 their parallel composition $S_1 \cdot S_2$ is also syntactically consistent.*

Definition 10 (A-closure) *For a set of actions A we define an A-closure of a pair of states $(s, t_1) \in states_S \times states_{T_1}$ as a subset Σ of $states_S \times states_{T_1}$ consisting of (s, t_1) itself and all pairs (s', t_1') in which s' can be reached from s by following a sequence of steps from $\longrightarrow_\Box^S$ labeled solely by actions in A and t_1' can be reached from t_1 by following an identical sequence (sequence with the same labels) of steps from $\longrightarrow_\Box^{T_1}$. Closures for pairs of states of S and T_2 are defined analogously.*

Definition 11 (A-closure) *We lift definition 10 to sets of pairs of states, such that the result is simply the union of the A-closures of all pairs.*

Let $t \xrightarrow{A}_\Box{}^* t'$ mean that t' is reachable from t via a possible empty sequence of required transitions labeled by actions from a set A (possibly different actions).

We will define observational consistency for states of a single automata.

Definition 12 *Let T be a modal automaton and let $A \subseteq act_T$ be a set of actions. A relation $C \subseteq states_T \times states_T$ is an observational consistency relation with respect to A if for any pair of states $(t_1, t_2) \in C$ the following two properties hold:*

1° $\forall t_1'.$ *if* $t_1 \xrightarrow{A}_\Box{}^* t_1'$ *then* $\forall a \notin A. \forall t_1''. t_1' \xrightarrow{a}_\Box t_1''$ *implies* $\exists t_2'. t_2 \xrightarrow{a}_\Diamond t_2' \wedge (t_1'', t_2') \in C$.

2° $\forall t_2'.$ *if* $t_2 \xrightarrow{A}_\Box{}^* t_2'$ *then* $\forall a \notin A. \forall t_2''. t_2' \xrightarrow{a}_\Box t_2''$ *implies* $\exists t_1'. t_1 \xrightarrow{a}_\Diamond t_1' \wedge (t_1', t_2'') \in C$.

Two states are observationally consistent if there exists an observational consistency relation relating them. A set of states is said to be observationally consistent with respect to A if all possible pairs of states from the set are observationally consistent with respect to A.

An automaton T is observationally consistent with respect to A iff the set $\{start_T\}$ is an observationally consistent set.

Lemma 4 *Consistency is transitive in the following sense: for a consistency relation C if $(t_1, t_2) \in C$ and $(t_2, t_3) \in C$ then $(t_1, t_3) \in C$.*

Lemma 5 *Let S, T_1, T_2 be modal I/O automata and $S \leq_\mathrm{m} T_1 \cdot T_2$. If $s \in states_S$ and $t_2 \in states_{T_2}$ are observationally consistent states wrt to $act_{T_2} \setminus act_{T_1}$ then projections of $(act_{T_2} \setminus act_{T_1})$–closure$(s, t_2)$ on the first and second[2] component give*

[2]For the current version of the proof we only need to claim consistency when projected on the first component.

observationally consistent sets of states with respect to the same set of actions $act_{T_2} \setminus act_{T_1}$.

Similarly if $s \in states_S$ *and* $t_1 \in states_{T_1}$ *are observationally consistent states wrt to* $act_{T_1} \setminus act_{T_2}$ *then projections of* $(act_{T_1} \setminus act_{T_2})$*–closure*$(s, t_1)$ *on the first and second component give observationally consistent sets of states with respect to the same set of actions* $act_{T_1} \setminus act_{T_2}$.

These claims generalize also to sets of consistent states.

Proof 4 (of Thm. 5) *We shall construct* S_1 *and* S_2 *exhibiting the requirements of the theorem. The signatures of* S_1 *and* S_2 *are identical to those of* T_1 *and* T_2*:*

$$int_{S_i} = int_{T_i}, \quad out_{S_i} = out_{T_i}, \quad int_{S_i} = int_{T_i} \ . \tag{C.8}$$

Since $S \leq_{\mathrm{m}} T_1 \cdot T_2$ *there exists the least relation* $R \subseteq states_S \times (states_{T_1} \times states_{T_2})$*, which is a modal refinement of* $T_1 \cdot T_2$ *by* S*. Let*

$$states_{S_1} = \{(\Sigma_1, t_1) \mid t_1 \in states_{T_1} \text{ and } \Sigma_1 \subseteq \{(s, t_2) \mid (s, (t_1, t_2)) \in R\}\} \tag{C.9}$$
$$states_{S_2} = \{(\Sigma_2, t_2) \mid t_2 \in states_{T_2} \text{ and } \Sigma_2 \subseteq \{(s, t_1) \mid (s, (t_1, t_2)) \in R\}\} \tag{C.10}$$

and

$$start_{S_1} = (\Sigma_1^0, start_{T_1}), \text{ where } \Sigma_1^0 = (act_{T_2} \setminus act_{T_1})\text{–closure}(start_S, start_{T_2}) \tag{C.11}$$
$$start_{S_2} = (\Sigma_2^0, start_{T_2}), \text{ where } \Sigma_2^0 = (act_{T_1} \setminus act_{T_2})\text{–closure}(start_S, start_{T_1}) \tag{C.12}$$

We create only one transition relation for each of S_1 *and* S_2 *(or more precisely both will have two, but identical transition relations). Intuitively this transition relation for* S_1 *will contain all steps allowed by* T_1 *and required by* S*. Formally it is given by the following rules:*

$$\frac{\begin{array}{c} a \in act_{S_1} \setminus act_{S_2} \qquad t_1 \xrightarrow{a}{}^{T_1}_{\Diamond} t_1' \qquad \exists (s, t_2) \in \Sigma_1 .\ s \xrightarrow{a}{}^{S}_{\Box} \\ \Sigma_1' = \{(s', t_2) \mid \exists (s, t_2) \in \Sigma_1 .\ s \xrightarrow{a}{}^{S}_{\Diamond} s'\} \end{array}}{(\Sigma_1, t_1) \xrightarrow{a}{}^{S_1}_{\Diamond} ((act_{T_2} \setminus act_{T_1})\text{–closure}(\Sigma_1'), t_1')} \tag{C.13}$$

$$\frac{\begin{array}{c} a \in act_{S_1} \cap act_{S_2} \qquad t_1 \xrightarrow{a}{}^{T_1}_{\Diamond} t_1' \qquad \exists (s, t_2) \in \Sigma_1 .\ s \xrightarrow{a}{}^{S}_{\Box} \\ \Sigma_1' = \{(s', t_2') \mid \exists (s, t_2) \in \Sigma_1 .\ s \xrightarrow{a}{}^{S}_{\Diamond} s' \wedge t_2 \xrightarrow{a}{}^{T_2}_{\Diamond} t_2'\} \end{array}}{(\Sigma_1, t_1) \xrightarrow{a}{}^{S_1}_{\Diamond} ((act_{T_2} \setminus act_{T_1})\text{–closure}(\Sigma_1'), t_1')} \tag{C.14}$$

$$\frac{a \in act_{S_1} \cap act_{S_2} \qquad t_1 \xrightarrow{a}{}^{T_1}_{\Box} t_1' \qquad \forall (s, t_2) \in \Sigma_1 .\ s \not\xrightarrow{a}{}^{S}_{\Box}}{(\Sigma_1, t_1) \xrightarrow{a}{}^{S_1}_{\Diamond} (\emptyset, t_1')} \tag{C.15}$$

$$\frac{a \in act_{S_1} \setminus act_{S_2} \qquad t_1 \xrightarrow{a}{}^{T_1}_{\Box} t_1'}{(\emptyset, t_1) \xrightarrow{a}{}^{S_1}_{\Diamond} (\emptyset, t_1')} \tag{C.16}$$

We take the must transition relation $\longrightarrow_{\Box}^{S_1}$ *to be identical with* $\longrightarrow_{\Diamond}^{S_1}$. *Note that effectively* S_1 *follows all must transition relations of* S *in its sort, except that whenever* T_1 *requires an input that is not followed by* S *(as* T_2 *is not able to synchronize on this input), we redirect the transition relation to a region where all must transitions of* T_1 *are mapped. We do that as minimum addition to maintain refinement of* T_1 *by* S_1*, on the functionality not explored by* S.

We refrain from showing the rules for S_2 *here—they can be easily constructed by analogy, as the problem is entirely symmetric.*

It is clear that the constructed systems S_1 *and* S_2 *are deterministic—the closure operation is deterministic and we apply to a unique maximal set for each action in each particular source state.*

Lemma 6 *The rules for transitions of* S_1 *ensures that if the originating state belongs to* $states_{S_1}$ *then the target state will also belong to* $states_{S_1}$.

An entirely symmetric lemma can be made for S_2.

Proof 5 *(Lemma 6) First we need to argue that the initial state* $start_{S_1} \in states_{S_1}$. *Firstly* $start_{T_1} \in states_{T_1}$ *which satisfies the first part of the requirement for states in* $states_{S_1}$. *Now we need to show that* $(act_{T_2} \setminus act_{T_1})$*–closure*$(start_S, start_{T_2}) \subseteq \{(s, t_2) \mid (s, (t_1, t_2)) \in R\}$. *The state from which the closure is calculated namely,* $(\{(start_S, start_{T_2})\}, start_{T_1})$, *is part of* $states_{S_1}$ *because* $(start_{S_1}, (start_{T_1}, start_{T_2})) \in R$. *All the transitions that are taken in the calculation of the closure are on actions not involving* T_1 *and are taken simultaneously by* S *and* T_2, *which ensures that all pairs of states* Σ_1' *that are reached will still fulfill the requirement for being in* $states_{S_1}$.

The rest of the proof consists of four cases, one for each rule. We need to argue for transitions generated by each of the four rules that the target state will be in $states_{S_1}$, *given that the source state is. Transitions generated by rule (C.13) ensure this because the states that are in* Σ_1' *have taken one transition that is on a non shared action of* T_1. *This transition is taken simultaneously by* T_1 *and* S. *Finally the closure also preserves the property, by the same argument as before. The argument for rule (C.14) is similar, the only difference being that the first transition is on a shared action and is taken by* S, T_1 *and* T_2. *Rule (C.15) and (C.16) are different. Here the argument is that* $\emptyset$ *is a subset of* $\{(s, t_2) \mid (s, (t_1, t_2)) \in R\}$.

We want to show that 1° $S_1 \leq_{\mathrm{m}} T_1$, 2° $S_2 \leq_{\mathrm{m}} T_2$ *and* 3° $S_1 \otimes S_2 \leq_{\mathrm{m}} S$.

1° *Show that*

$$R_1 = \{((\Sigma_1, t_1), t_1) \mid \Sigma_1 \in states_{S_1} \text{ and } t_1 \in states_{T_1}\} \tag{C.17}$$

is a modal refinement of T_1 *by* S_1.

Consider an arbitrary pair of states $((\Sigma_1, t_1), t_1) \in R_1$ *and a transition* $t_1 \xrightarrow{a}{}^{T_1}_{\Box} t_1'$. *We want to show that there exists a state* (Σ_1', t_1') *and a transition such that* $(\Sigma_1, t_1) \xrightarrow{a}{}^{S_1}_{\Box} (\Sigma_1', t_1')$ *and* $((\Sigma_1', t_1'), t_1') \in R_1$

1.1° *If* $\Sigma_1 = \emptyset$ *then take* Σ_1' *to be* $\emptyset$ *and the corresponding transition exists due to rule (C.16) or rule (C.15). In the case of rule (C.15) the premise that* $\forall (s, t_2) \in \Sigma_1$ *is trivially true.*

1.2° *Let* a *be an action of* T_1 *that is not shared with* T_2, *or similarly* $a \in act_{S_1} \setminus act_{S_2}$. *We want to apply rule (C.13) and want to show that the premises are fulfilled. The first two premises are fulfilled by the case that we are looking at. The third premise is fulfilled by the following argument. Because* t_1' *is making a step we have that* $(t_1, t_2) \xrightarrow{a}{}^{T_1 \cdot T_2}_{\Box} (t_1', t_2)$. *By the definition of* $states_{S_1}$ *and* R_1 *we have that* $(s, (t_1, t_2)) \in R$ *for every pair* $(s, t_1) \in \Sigma_1$. *Because* R *is a modal refinement of* T *by* S *we have that* $s \xrightarrow{a}{}^{S}_{\Box} s'$ *and* $(s', (t_1', t_2)) \in R$ *for every pair* $(s, t_1) \in \Sigma_1$. *The third premise will trivially hold and we can even conclude that* Σ_1' *will be nonempty. Now we can apply rule (C.13) and we can conclude that indeed* $(\Sigma_1, t_1) \xrightarrow{a}{}^{S_1}_{\Diamond} (((act_{T_2} \setminus act_{T_1})\text{–}closure(\Sigma_1'), t_1')$. *From this we can conclude that a similar must transition exists because the two transition relations are identical. Finally we can conclude that* $(((act_{T_2} \setminus act_{T_1})\text{–}closure(\Sigma_1'), t_1') \in R_1$ *because the generated transitions stay within* $states_{S_1}$ *and* $t_1' \in states_{T_1}$.

1.3° *Let* a *be an action of* T_1 *that is shared with* T_2, *or similarly* $a \in act_{S_1} \cap act_{S_2}$. *We want to apply rule (C.14) and (C.15), in two different sub cases, and want to show that the premises are fulfilled. The first two premises of both rules are fulfilled by the case that we are looking at. The third premise of rule (C.14) and (C.15) are each others opposites, such that the one is true when the other is false and vise versa. Looking at the case where* $\exists (s, t_2) \in \Sigma_1 . s \xrightarrow{a}{}^{S}_{\Box}$, *which is exactly the third premise of rule (C.14), then we can conclude that the last premise for rule (C.14) is true by the following argument. Because* S *is consistent we know that there is a transition* $s \xrightarrow{a}{}^{S}_{\Diamond}$. *Because* R *is a modal refinement of* T *by* S *and we can conclude that the only way that this transition can exist is if a similar transition* $t_2 \xrightarrow{a}{}^{T_2}_{\Diamond} t_2'$ *exists such that* $(t_1, t_2) \xrightarrow{a}{}^{T_1 \cdot T_2}_{\Diamond}$. *The fourth premise of rule (C.14) is trivially true, but we can now conclude that* Σ_1' *is nonempty. Now we can apply rule (C.14) and we can conclude that indeed* $(\Sigma_1, t_1) \xrightarrow{a}{}^{S_1}_{\Diamond} (((act_{T_2} \setminus act_{T_1})\text{–}closure(\Sigma_1'), t_1')$. *From this we can conclude that a similar must transition exists because the two transition relations are identical. Finally we can conclude that* $(((act_{T_2} \setminus act_{T_1})\text{–}closure(\Sigma_1'), t_1') \in R_1$ *because the generated transitions stay within* $states_{S_1}$ *and* $t_1' \in states_{T_1}$.

Now turning to the other sub case where $\forall (s,t_2) \in \Sigma_1\ s \not\xrightarrow{a}{}^{S}_{\Box}$. *In this case there are no must transitions in* S *requiring the behavior but* S_1 *will have the behavior because* T_1 *requires it. From this we can conclude that* $(\emptyset,t_1) \xrightarrow{a}{}^{S_1}_{\Diamond} (\emptyset,t_1')$ *and that a similar must transition exists because the two transition relations are identical. Finally we can conclude that* $(((act_{T_2} \setminus act_{T_1})\text{–}closure(\emptyset), t_1') \in R_1$ *because the generated transitions stay within* $states_{S_1}$ *and* $t_1' \in states_{T_1}$.

This finishes one direction of the proof. Lets now consider a may transition $(\Sigma_1,t_1) \xrightarrow{a}{}^{S_1}_{\Diamond} (\Sigma_1',t_1')$. *We need to show that a transition* $t_1 \xrightarrow{a}{}^{T_1}_{\Diamond} t_1'$ *exists such that* $((\Sigma_1',t_1') \in R_1)$

1.4° *This transition could have been generated by one of the four rules (C.13)-(C.16). In two of the cases we can directly conclude that a transition* $t_1 \xrightarrow{a}{}^{T_1}_{\Diamond} t_1'$ *exists. In the other two cases we can conclude that this transition exists because the rules require a similar must transition and* T_1 *is syntacticly consistent. Now it follows directly from Lemma 6 that* $(\Sigma_1',t_1') \in R_1$

2° *The proof that* $S_2 \leq_{\mathrm{m}} T_2$ *is entirely symmetric to the proof that* $S_1 \leq_{\mathrm{m}} T_1$.

3° *Show that* $S_1 \otimes S_2 \leq_{\mathrm{m}} S$. *We do that by arguing that*

$$\begin{aligned} R_2 = \{(((\Sigma_1,t_1),(\Sigma_2,t_2)),s) \mid \\ ((act_{T_1} \setminus act_{T_2})\text{–}closure(s,t_1) \subseteq \Sigma_2 \text{ and} \\ ((act_{T_2} \setminus act_{T_1})\text{–}closure(s,t_2) \subseteq \Sigma_1 \text{ and} \\ \Pi_1(\Sigma_1) \text{ is observationally consistent wrt } act_{T_2} \setminus act_{T_1} \text{ and} \\ \Pi_1(\Sigma_2) \text{ is observationally consistent wrt } act_{T_1} \setminus act_{T_2}\} \end{aligned} \tag{C.18}$$

is a modal refinement of S *by* $S_1 \otimes S_2$. *First we should argue that*

$$((start_{S_1}, start_{S_2}), start_S) \in R_2\ . \tag{C.19}$$

Obviously

$$(act_{T_2} \setminus act_{T_1})\text{–}closure(start_S, start_{T_2}) \subseteq \Sigma_1^0 \text{ and} \tag{C.20}$$

$$(act_{T_1} \setminus act_{T_2})\text{–}closure(start_S, start_{T_1}) \subseteq \Sigma_2^0 \tag{C.21}$$

(actually equalities hold). Observational consistency of projections of Σ_1^0 *and* Σ_2^0 *follows from consistency of* S, T_1, T_2 *and Lemma 5.*

We shall discuss that the may transition relation preserves the refinement. Take any $(((\Sigma_1,t_1),(\Sigma_2,t_2)),s) \in R_2$ *and a transition step*

$$((\Sigma_1,t_1),(\Sigma_2,t_2)) \xrightarrow{a}{}^{S_1 \otimes S_2}_{\Diamond} ((\Sigma_1',t_1'),(\Sigma_2',t_2')) \tag{C.22}$$

We want to find a state s' such that $s \xrightarrow{a}{}^{S}_{\Diamond} s'$ and $((\Sigma_1', t_1'), (\Sigma_2', t_2')), s') \in R_2$.

Note that due to the way R_2 is constructed we know that neither Σ_1 nor Σ_2 are empty. The transition step of the composition must then be created by both components taking a shared action (and both following rule (C.14)) or by one component taking a non-shared action, by rule (C.13), and the other not changing state.

Observe that rule (C.16)*, can never give rise to such a transition as it would require Σ_1 or Σ_2 to be empty, which we have just ruled out.*

3.1° *Let $a \in act_{S_1} \cap act_{S_2}$. We want to first argue that both components take steps generated by rule* (C.14) *and not rule* (C.15)*. The latter would require that either t_1 or t_2 enjoys a must transition $t_i \xrightarrow{a}{}^{T_i}_{\Box} t_i'$. If both transitions existed, they would imply that also $s \xrightarrow{a}{}^{S}_{\Box} s'$ (since $(s, (t_1, t_2)) \in R$, S is deterministic), which would contradict the joint premises of the rules. So only one of the two must transitions can exist. But then the other component is taking a transition generated by rule* (C.14) *implying that $s \xrightarrow{a}{}^{S}_{\Box} s'$, contradicting premises of rule* (C.15) *(for both components). In other words rule* (C.15) *could not have been used, so for some sets Σ_1'', Σ_2'':*

$$(\Sigma_1, t_1) \xrightarrow{a}{}^{S_1}_{\Diamond} ((act_{T_2} \setminus act_{T_1})\text{–}closure(\Sigma_1''), t_1') \tag{C.23}$$

$$(\Sigma_2, t_2) \xrightarrow{a}{}^{S_2}_{\Diamond} ((act_{T_1} \setminus act_{T_2})\text{–}closure(\Sigma_2''), t_2') \tag{C.24}$$

From that we derive that rule (C.14) *must have been used to create both of these transitions, which implies that there exists $(s_1, p_2) \in \Sigma_1$ such that $s_1 \xrightarrow{a}{}^{S}_{\Box} s_1'$ for some state s_1'. Since $\Pi_1(\Sigma_1)$ is an observationally consistent set with respect to $act_{T_2} \setminus act_{T_1}$ then there exists a state s' such that $s \xrightarrow{a}{}^{S}_{\Diamond} s'$ and (s_1', s') is an observationally consistent pair of states. Since S is deterministic the same argument can be used for all elements in $\Pi_1(\Sigma_1'')$[3], which with help of Lemmas 4 and 5 leads us to a conclusion that the first component of $(act_{T_2} \setminus act_{T_1})$–$closure(\Sigma_1'')$ is observationally consistent wrt $(act_{T_2} \setminus act_{T_1})$.*

Since rule (C.14)*, or more precisely its counterpart for S_2, must have been used to construct transition* (C.24) *we can also conclude that $t_2 \xrightarrow{a}{}^{T_2}_{\Diamond} t_2'$. So by premises of rule* (C.14) *instantiated for transition* (C.23) *we conclude that $(s', t_2') \in \Sigma_1''$ and hence is in the closure.*

Symmetric arguments can be used to argue that the first component of the closure of Σ_2'' is observationally consistent wrt $act_{T_1} \setminus act_{T_2}$, and

[3]In the nondeterministic case we would probably have to extend the definition of observational consistency with a universal quantifier, instead of the existential, which it is using now.

that $(s', t'_1) \in \Sigma''_2$ *and hence also in its closure, which finishes the proof of this case.*

3.2° *Let* $a \in act_{S_1} \setminus act_{S_2}$. *Then we know that:*

$$(\Sigma_1, t_1) \xrightarrow{a}{}^{S_1}_{\diamond} (\Sigma'_1, t'_1) \text{ and } \Sigma'_2 = \Sigma_2 \text{ and } t'_2 = t_2 \ . \tag{C.25}$$

It easy to conclude that the step of T_1 *has been generated by rule* (C.13) *and not rule* (C.16) *(we have already argued against this case above:* $\Sigma_1 \neq \emptyset$*).*

The fact that (Σ_1, t_1) *is able to make an a step by rule* (C.13) *implies that some state of* s *paired with some state of* T_2 *in* Σ_1 *requires such a step. By observational consistency of* $\Pi_1(\Sigma_1)$ *we have that necessarily* $s \xrightarrow{a}{}^{S}_{\diamond} s'$ *for some* s'. *Moreover* $(s', t_2) \in \Sigma'_1$ *(by rule* (C.13)*) and* $(s', t'_1) \in \Sigma_2$ *since* $(s', t'_1) \in (act_{T_1} \setminus act_{T_2})$*–closure*$(s, t_1) = \Sigma_2$. *Since* Σ_2 *does not change, there is no need to argue for its consistency. Consistency of* $\Pi_1(\Sigma'_1)$ *follows from the fact that a transition is taken, which cannot move outside the consistent set (a hidden must transition).*

3.3° *The case when* s *takes a transition over a non-shared action of* S_2 *is entirely symmetric.*

Observe that implicitly (by analyzing all interaction possibilities) we have ruled out a possibility of a deadlock between S_1 *and* S_2.

Let us now turn towards the must transition relations. Assume that for some action a *and state* s' *we have that* $s \xrightarrow{a}{}^{S}_{\Box} s'$.

3.4° *Let* $a \in act_{T_1} \cap act_{T_2}$. *Since* $(s, (t_1, t_2)) \in R$ *and* S *is syntactically consistent, we get that* $(t_1, t_2) \xrightarrow{a}{}^{T_1 \cdot T_2}_{\diamond} (t'_1, t'_2)$ *for some* t'_1, t'_2 *and further that* $t_1 \xrightarrow{a}{}^{T_1}_{\diamond} t'_1$ *and* $t_2 \xrightarrow{a}{}^{T_2}_{\diamond} t'_2$. *But these imply by rule* (C.14) *that* $(\Sigma_1, t_1) \xrightarrow{a}{}^{S_1}_{\diamond} (\Sigma'_1, t'_1)$, *where* $(act_{T_2} \setminus act_{T_1})$*–closure*$(s', t'_2) \subseteq \Sigma'_1$ *and similarly* $(\Sigma_2, t_2) \xrightarrow{a}{}^{S_2}_{\diamond} (\Sigma'_2, t'_2)$, *where* $(act_{T_1} \setminus act_{T_2})$*–closure*$(s', t'_1) \subseteq \Sigma'_2$.

We have chosen that the must transition relations of both S_1 *and* S_2 *are identical with their respective may transition relations, so we can conclude that* $((\Sigma_1, t_1), (\Sigma_2, t_2)) \xrightarrow{a}{}^{S_1 \otimes S_2}_{\Box} ((\Sigma'_1, t'_1), (\Sigma'_2, t'_2))$.

Observational consistency of the first components of Σ'_1 *and* Σ'_2 *can be argued as in earlier cases (existence of a single must transition of* s *guarantees that none of* s *transitions labeled in* a *and sourced in states of* Σ_i *can leave outside the set of consistent states).*

3.5° *Let* $a \in act_{T_1} \setminus act_{T_2}$. *Since* $(s, (t_1, t_2)) \in R$ *and* S *is syntactically consistent, we get that* $(t_1, t_2) \xrightarrow{a}{}^{T_1 \cdot T_2}_{\diamond} (t'_1, t_2)$ *and further that* $t_1 \xrightarrow{a}{}^{T_1}_{\diamond} t'_1$. *But this implies by rule* (C.13) *that* $(\Sigma_1, t_1) \xrightarrow{a}{}^{S_1}_{\diamond} (\Sigma'_1, t'_1)$, *where* $(act_{T_2} \setminus$

act_{T_1})–*closure*$(s', t_2) \subseteq \Sigma_1'$. *Also* $(act_{T_1} \setminus act_{T_2})$–*closure*$(s', t_1') \subseteq \Sigma_2$ *since the transition performed by this pair is within the original closure* $(act_{T_1} \setminus act_{T_2})$–*closure*$(s, t_1)$, *which was a subset of* Σ_2.

As we have chosen that must transition relation of S_1 *is identical with its may transition relation, we can conclude that:*

$$((\Sigma_1, t_1), (\Sigma_2, t_2)) \xrightarrow{a}{}^{S_1 \otimes S_2}_{\Box} ((\Sigma_1', t_1'), (\Sigma_2, t_2)) \ . \tag{C.26}$$

Finally Σ_1' *is observationally consistent as* s *only takes a hidden transition here (with respect to the set of ignored actions), which finishes the proof for this case.*

3.6° *The case where* S_2 *takes an independent step is symmetric.*□

Observe that the above theorem can be used to generate decompositions of simulations and bisumulations (which are special cases of modal refinement).

Proof 6 (Thm. 6) *Show that*

$$R_3 = \{((i_1, i_2), (t_1, t_2)) \in states_{I_1 \cdot I_2} \times states_{T_1 \cdot T_2} \mid i_1 \leq_{\mathrm{m}} t_1 \wedge i_2 \leq_{\mathrm{m}} t_2\} \tag{C.27}$$

is a modal refinement of $T_1 \cdot T_2$ *by* $I_1 \cdot I_2$.

1° *Consider* $(i_1, i_2) \xrightarrow{a}_{\diamond} (i_1', i_2')$. *We have to consider four cases:* 1.1° $a \in ext_{I_1 \cdot I_2}$, $i_1 \xrightarrow{a}_{\diamond} i_1'$ *and* $i_2 = i_2'$. *As* $i_1 \leq_{\mathrm{m}} t_1$ *there must exist a* t_1' *such that* $t_1 \xrightarrow{a}_{\diamond} t_1'$ *and* $i_1' \leq_{\mathrm{m}} t_1'$, *so* $((i_1', i_2), (t_1', t_2)) \in R_3$. *By definition of the composition operator* $(\cdot)$ *we get that* $(t_1, t_2) \xrightarrow{a}_{\diamond} (t_1', t_2)$*: the only possibility for it could not hold is when* (t_1', t_2) *has been pruned in* $T_1 \cdot T_2$, *so there exists a sequence of internally controllable must transitions leading from* (t_1', t_2) *to an error state* (t_1'', t_2'') *where* $t_k'' \xrightarrow{a!}_{\diamond} t_k'''$ *and* $t_{3-k}'' \not\xrightarrow{a?}_{\diamond}$, *where* $k \in 1, 2$. *But then a corresponding sequence would exist in* $I_1 \cdot I_2$, *meaning that* $(i_1, i_2) \xrightarrow{a}_{\diamond} (i_1', i_2)$ *was not possible to begin with (also pruned). Finally it is easy to see* $((s_1', s_2'), (t_1', t_2')) \in R_3$.

1.2° $a \in ext_{I_1 \cdot I_2}$, $i_2 \xrightarrow{a}_{\diamond} i_2'$ *and* $i_1 = i_1'$ *is symmetric.*

1.3° $a \in int_{I_1 \cdot I_2}$, $i_1 \xrightarrow{a!}_{\diamond} i_1'$ *and* $i_2 \xrightarrow{a?}_{\diamond} i_2'$. *Then by* $i_1 \leq_{\mathrm{m}} t_1$ *and* $i_2 \leq_{\mathrm{m}} t_2$ *we conclude that there exists* t_1', t_2' *such that* $t_1 \xrightarrow{a!}_{\diamond} t_1'$ *and* $t_2 \xrightarrow{a?}_{\diamond} t_2'$ *and* $i_1' \leq_{\mathrm{m}} t_1'$ *and* $i_2' \leq_{\mathrm{m}} t_2'$. *By definition of the composition operator* $(\cdot)$ *we get that* $(t_1, t_2) \xrightarrow{a}_{\diamond} (t_1', t_2')$*: the only possibility for it could not hold is when* (t_1', t_2') *has been pruned in* $T_1 \cdot T_2$, *so there exists a sequence of internally controllable must transitions leading from* (t_1', t_2') *to an error state* (t_1'', t_2'') *where* $t_k'' \xrightarrow{a!}_{\diamond} t_k'''$ *and* $t_{3-k}'' \not\xrightarrow{a?}_{\diamond}$, *where* $k \in 1, 2$. *But then a corresponding sequence would exist in* $I_1 \cdot I_2$, *meaning that* $(i_1, i_2) \xrightarrow{a}_{\diamond} (i_1', i_2')$ *was not possible to begin with. Finally it is easy to see* $((s_1', s_2'), (t_1', t_2')) \in R_3$.

1.4° $a \in int_{I_1 \cdot I_2}$, $i_2 \xrightarrow{a!}_{\diamond} i_2'$ *and* $i_1 \xrightarrow{a?}_{\diamond} i_1'$. *The argument follows as in* 1.3°.

2° *Consider* $(t_1, t_2) \xrightarrow{a}_{\Box} (t_1', t_2')$. *We have four subcases again out of which 2 are interesting.*

2.1° $a \in ext_{T_1 \cdot T_2}$ *and* $t_1 \xrightarrow{a}_{\Box} t'_1$ *and* $t_2 = t'_2$. *Then by* $i_1 \leq_{\mathrm{m}} t_1$ *there exist* i'_1 *such that* $i_1 \xrightarrow{a}_{\Box} i'_1$ *and* $i'_1 \leq_{\mathrm{m}} t'_1$. *By similar argument as above* $(i_1, i_2) \xrightarrow{a}_{\Box} (i'_1, i_2)$ *(because if* (i'_1, i_2) *was pruned then so was* (i_1, i_2)*, for which we assumed that it was not) and* $(i'_1, i'_2), (t'_1, t'_2) \in R_3$.

2.2° $a \in ext_{T_1 \cdot T_2}$ *and* $t_2 \xrightarrow{a}_{\Box} t'_2$ *and* $t_1 = t'_1$. *Argument as above.*

2.3° $a \in int_{T_1 \cdot T_2}$ *and* $t_1 \xrightarrow{a!}_{\Box} t'_1$ *and* $t_2 \xrightarrow{a?}_{\Box} t'_2$. *Then by* $i_1 \leq_{\mathrm{m}} t_1$ *and* $i_2 \leq_{\mathrm{m}} t_2$ *there exist* i'_1 *and* i'_2 *such that* $i_1 \xrightarrow{a}_{\Box} i'_1$ *and* $i_2 \xrightarrow{a}_{\Box} i'_2$ *and* $i'_1 \leq_{\mathrm{m}} t'_1$ *and* $i'_2 \leq_{\mathrm{m}} t'_2$. *By a similar argument involving the definition of* $(\cdot)$ *as above we get* $(i_1, i_2) \xrightarrow{a}_{\Box} (i'_1, i'_2)$ *(as if* (i'_1, i'_2) *then so would* (i_1, i_2) *which was assumed not to be pruned). So* $((i'_1, i'_2), (t'_1, t'_2)) \in R_3$*, which finishes the proof.*□

Paper D

On Modal Refinement and Consistency

Kim G. Larsen, Ulrik Nyman
Department of Computer Science,
Aalborg University, Denmark

Andrzej Wąsowski
Computational Logic and Algorithms Group,
IT University of Copenhagen, Denmark

Abstract

Almost 20 years after the original conception, we revisit several fundamental question about modal transition systems. First, we demonstrate the incompleteness of the standard modal refinement using a counterexample due to Hüttel. Deciding any refinement, complete with respect to the standard notions of implementation, is shown to be computationally hard (co-NP hard). Second, we consider four forms of consistency (existence of implementations) for modal specifications. We characterize each operationally, giving algorithms for deciding, and for synthesizing implementations, together with their complexities.

Keywords: Modal Transition Systems, Modal Refinement, Consistency, Operational Characterization, Synthesizing Implementations

1 Background and Overview

Modal transition systems (MTSs) are a generalization of labeled transition systems (LTSs). Similarly to LTSs modal transition systems use labeled transitions between states to model behaviors. Unlike LTSs, they distinguish allowed and required behaviors (over- and under-approximations), which makes them a suitable semantic model for abstraction in program analysis and verification.

MTSs, originally introduced by Larsen and Thomsen almost 20 years ago [LT88], have since been applied in program analysis [HJS01, Sch01], model checking [GHJ01, BLS93], verification [**?**, Bru97], equation solving [LX90], interface theories [LNW07a] (Paper C), software product lines [LNW07a, FUB06] and model merging [UC04, BCU06]. Foundational work on modal transition systems included extensions to modal hybrid systems [CD97], timed modal specifications [Lar89, ČGL93, LSW96] and variants of disjunctive MTSs [LX90, FH06, SF07]. Surprisingly though, several fundamental questions about the theory of MTSs have never been addressed.

Refinement relations for modal transition systems are defined contravariantly. If S refines T then all allowed behaviors of S need to be allowed in T, while all required behaviors of T need also be required by S. An *implementation* is an MTS that has been completely specified, i.e. all its allowed behavior is also required, leaving no further choice for refinement. One fundamental issue for a modal refinement is to see whether it characterizes the inclusion of implementation sets thoroughly: can one for an MTS S refining an MTS T imply that all implementations of S are also implementations of T? And vice-versa?

Standard modal refinement is sound, but not complete in this sense. Meaning that here exist MTSs for which implementation inclusion holds, but which do not refine each other. We show that deciding any sound and complete refinement, preserving the set of implementations of standard modal refinement or weak modal refinement is co-NP hard. We conjecture the same for may-weak modal refinement [LNW07a] (Paper C) and branching refinement [FUB06].

Modal transition systems of [LT88] are *syntactically consistent*, meaning that any required transition must also be allowed. This effectively disallows reasoning about inconsistencies, which is necessary for proper treatment of logical connectives in the context of modal transition systems (for example one would like to be able to express a modal transition system expressing a conjunction of two other MTSs that represent contradictory specifications). On the other hand, in [LNW07a] (Paper C), we have observed that other, more behavioral, notions of consistency might be useful. We have shown that systems that are *observationally consistent* with respect to some set of hidden actions, can be decomposed using parallel decomposition. We used this observation to build a product line theory in which modal transition systems play the role of behavioral variability models.

We believe that consistency should be decoupled from the basic definition of

a modal transition system. In our opinion understanding a notion of consistency requires relating it to a notion of satisfiability, as typically done in logics. For example: a propositional formula is consistent if there exists a truth assignment on which the formula evaluates to true. In our context, modal transition systems play the role of formulæ, truth assignments are concrete implementations, and a refinement preorder is our satisfaction relation. Consequently, instead of proposing ad hoc *criteria* for consistency, we define consistency of a specification *semantically* as existence of a concrete implementation refining it.

Altogether we discuss four modal refinements and their induced consistencies. For each of these we define consistency semantically and find a computable criterion (a consistency relation) for deciding it. Then we study the complexity of consistency and the criterion. The results are summarized in Table D.1.

Our choice of refinements and consistencies for this study is driven by existing work. We choose one known consistency (syntactic consistency) that have not been characterized using a refinement, and three known refinements (strong, may-weak and weak modal refinement) for which the related notions of consistency had never been formulated. However, we believe that consistency is not only of theoretical interest. Inconsistencies in specifications typically indicate modeling errors and thus procedures for detecting them find use in tools.

The contents of this paper are: the definition of modal transition systems and their refinement (Section 2), complexity analysis of completeness of this refinement (Section 3), a discussion of consistency notions induced by four modal refinements (Sections 4–7), a summary and a list of open problems (Section 8).

2 Modal Transition Systems

We introduce the basics following Larsen and Thomsen [LT88]. Assume a global set of actions act and write act^τ for $act \cup \{\tau\}$, where τ is a distinct internal action, such that $\tau \notin act$. A modal transition system is a triple $S = (states_S, \longrightarrow^S, \dashrightarrow^S)$, where $states_S$ is a set of states, also known as specifications [LT88] or processes. Then $\longrightarrow^S \subseteq states_S \times act^\tau \times states_S$ is a must-transition relation representing required transitions, and $\dashrightarrow^S \subseteq states_S \times act^\tau \times states_S$ is a may-transition relation representing allowed transitions.

In general the sets of states and transitions may be infinite, but we restrict ourselves to finite state systems with finite sets of actions in this paper. For

Table D.1: Summary of consistency-related results.

Modal refinement	Consistency	Lower bound	Upper bound	Section
syntactic	syntactic consistency [LT88]	linear time	linear time	4
strong [LT88]	strong consistency	NP-hard	exponential time	5
weak [HL89]	weak consistency	NP-hard	exponential time	6
may-weak [LNW07a]	may-weak consistency	NP-hard	exponential time	7

simplicity we write $s \xrightarrow{a}{}^{S} s'$ iff $(s, a, s') \in \longrightarrow^{S}$, and $s \dashrightarrow^{a,S} s'$ iff $(s, a, s') \in \dashrightarrow^{S}$.

Larsen and Thomsen originally designed modal transition systems to be syntactically consistent meaning that all required transitions are also allowed: $\longrightarrow^{S} \subseteq \dashrightarrow^{S}$. Already in [Lar89] Larsen lifts this restriction, with the argument that any sufficiently expressive specification language needs to be able to specify inconsistent specifications. This means that our transition systems are very much like mixed transition systems of Dams [Dam96]. In Section 3 we follow the syntactic consistency requirement, while we relax it in later sections, generalizing the notion of consistency to strong and weak behavioral preorders. Regardless whether the consistency assumption is in place or not, we always separate the two transition relations explicitly to avoid confusion. A solid arrow represents just a must transition, without the possible related may transition. We draw both arrows when talking about a syntactically consistent must transition.

A modal transition system I is an *implementation* when the two transition relations coincide, $\longrightarrow^{I} = \dashrightarrow^{I}$. We use capital I to denote implementations and always state explicitly whenever a modal transition system is an implementation.

The following is the standard notion of strong refinement for modal transition systems introduced in [LT88] and generally accepted ever since:

Definition 1 (Modal Refinement) *For a pair of modal transition systems S and T a binary relation $\mathcal{R} \subseteq states_S \times states_T$ is a modal refinement between states of S and T iff for all $(s,t) \in \mathcal{R}$ and all actions a it holds that:*

for all $t' \in states_T$ such that $t \xrightarrow{a}{}^{T} t'$
there exists an $s' \in states_S$ such that $s \xrightarrow{a}{}^{S} s'$ and $(s', t') \in \mathcal{R}$,

for all $s' \in states_S$ such that $s \dashrightarrow^{a,S} s'$
there exists a $t' \in states_T$ such that $t \dashrightarrow^{a,T} t'$ and $(s', t') \in \mathcal{R}$.

We say that a state $s \in states_S$ refines a state $t \in states_T$, written $s \leq_{\mathrm{m}} t$, iff there exists a modal refinement containing (s,t).

If $\longrightarrow^{T} = \emptyset$ then this refinement collapses to regular simulation [HM85, Lar87], while it coincides with bisimulation equivalence [Par81, Mil83] if S and T are implementations.

3 Non-thoroughness of Modal Refinement

Already in the eighties there have been rumors of modal refinement being incomplete. However we were unable to find a published account of this fact, so we decided to include it here. We shall now define what we mean by completeness, proceeding to a counterexample witnessing the incompleteness of modal refinement. After this brief introduction we move to the first contribution of the paper: a discussion of the complexity class of a hypothetical complete refinement. For

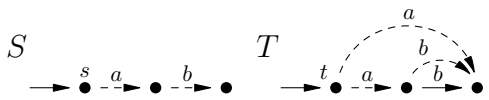


Figure D.1: $[\![S, s]\!] \subseteq [\![T, t]\!]$ and $s \not\leq_{\mathrm{m}} t$

a state $s \in \mathit{states}_S$ let $[\![S, s]\!]$ denote the set of all its implementations such that $[\![S, s]\!] = \{(I, i) \mid i \leq_{\mathrm{m}} s \text{ and } \longrightarrow^I = \dashrightarrow^I\}$. Modal refinement is known to be sound, with respect to implementation inclusion: for $s \in \mathit{states}_S \wedge t \in \mathit{states}_T$, if $s \leq_{\mathrm{m}} t$ then also $[\![S, s]\!] \subseteq [\![T, t]\!]$, which follows directly from transitivity of $\leq_{\mathrm{m}}$. However $\leq_{\mathrm{m}}$ is not complete in this sense: there exist specifications S and T, with states s, t, such that $[\![S, s]\!] \subseteq [\![T, t]\!]$ but $s \not\leq_{\mathrm{m}} t$. This property of modal refinement is sometimes known as non-thoroughness [GJ02]. Figure D.1 presents a counterexample originating in the thesis of Hüttel [Hüt88, p. 32], also found in the thesis of Xinxin [Xin92, p. 87] and in [SF07], albeit disguised in the context of disjunctive modal transition systems [LX90][1]. It contains two specifications S, T. It is a simple exercise to see that $[\![S, s]\!] = [\![T, t]\!]$, while $s \not\leq_{\mathrm{m}} t$.

3.1 A Thorough Refinement is Co-NP Hard

Despite the non-thoroughness (incompleteness) of modal refinement its usefulness has never been questioned. This is probably because modal refinement is a natural generalization of both simulation and bisimulation and because it can be established efficiently (in time polynomial in the size of the transition systems). By showing that any complete refinement preserving precisely the same set of implementations as $\leq_{\mathrm{m}}$ cannot be decided in polynomial time (unless P=NP), we give yet another argument in favor of $\leq_{\mathrm{m}}$.

We show co-NP hardness by reducing 3-DNF-TAUTOLOGY to checking a sound and complete modal refinement in the above sense. Consider a propositional formula φ over n variables $x_1, \ldots, x_n$. It is clear that φ is a tautology iff $\mathit{true} \Rightarrow \varphi$ is a tautology. We will show how to construct, in polynomial time, a modal transition systems T_φ (representing a tautology over $x_1 \ldots x_n$) and S_φ (representing φ), so that $\mathit{true} \Rightarrow \varphi$ is a tautology iff $[\![T_\varphi, \mathit{true}]\!] \subseteq [\![S_\varphi, \varphi]\!]$, for selected initial states true and φ of T_φ and S_φ respectively. For simplicity we will assume that all clauses of φ are satisfiable. Satisfiability of a clause consisting of three conjunctions can be decided in constant time. Unsatisfiable clauses can thus be removed from φ in polynomial time, before we construct T_φ and S_φ. We choose the following states and actions for S_φ:

$$\mathit{states}_{S_\varphi} = \{\varphi, c_1, \ldots, c_m, \mathbf{0}\} \quad \{a, x_1, \ldots, x_n\} \subseteq \mathit{act} \ , \tag{D.1}$$

[1] We thank Michael Huth, Harald Fecher, Heiko Schmidt and one of the anynonymous reviewers for helping to track down its origins.

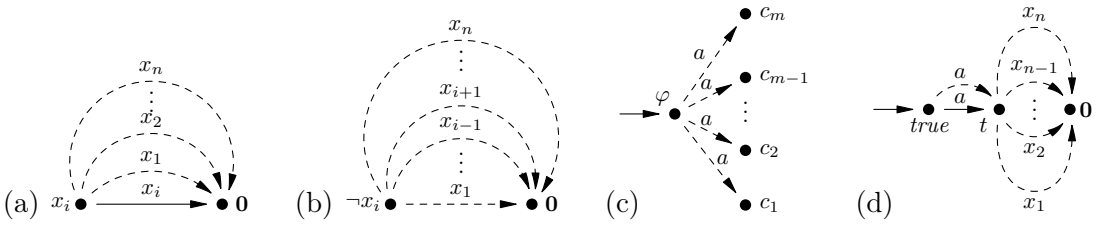


Figure D.2: Representing (a) a positive literal, (b) a negative literal, (c) a 3-DNF formula $\varphi = c_1 \vee \cdots \vee c_m$ and (d) a tautology over variables $x_1 \ldots x_n$.

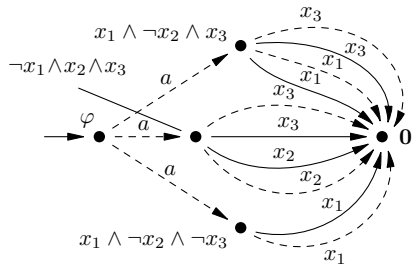


Figure D.3: Reduction for $\varphi = (x_1 \wedge \neg x_2 \wedge x_3) \vee (\neg x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3)$.

where c_i are clauses of φ, while $\mathbf{0}$ and a are fresh names.

First we explain how a single literal can be represented as a state with at most $n+1$ outgoing transitions. For a positive literal x_i we introduce a state x_i with a required transition $x_i \xrightarrow{x_i} \mathbf{0}$ and allowed transitions $x_i \dashrightarrow^{x_k} \mathbf{0}$ for all $k = 1..n$. For a negative literal $\neg x_i$ we allow no outgoing must transitions and create may transitions $(\neg x_i) \dashrightarrow^{x_k} \mathbf{0}$ for all $k \neq i$. Positive assignments are represented by must transitions, and negative assignments are represented by lack of may transitions. Assignments with no effect on satisfaction of the formula are modeled by may transitions with no corresponding must transitions. See Figure D.2ab.

Now generalize this to conjunctive clauses of a 3-DNF formula. A clause $l_1 \wedge l_2 \wedge l_3$ is translated into a state labeled $l_1 \wedge l_2 \wedge l_3$ with the following transitions:

1° $(l_1 \wedge l_2 \wedge l_3) \xrightarrow{x_i}{}^{S_\varphi} \mathbf{0}$ iff $l_k = x_i$ for some $k = 1..3$.

2° $(l_1 \wedge l_2 \wedge l_3) \dashrightarrow^{x_i}{}^{S_\varphi} \mathbf{0}$ iff $l_k \neq \neg x_i$ for all $k = 1..3$.

Since we only consider satisfiable clauses, modal transition systems created this way are syntactically consistent (all required transitions are allowed). A satisfying truth assignment to $l_1 \wedge l_2 \wedge l_3$ can be extracted from any implementation I refining the state with the same label—just set x_i to *true* iff $I \xrightarrow{x_i}$ and set x_i to *false* otherwise. Similarly we can construct an implementation refining $l_1 \wedge l_2 \wedge l_3$ given any satisfying assignment to this clause.

A 3-DNF formula $\varphi = c_1 \vee \ldots \vee c_m$ is represented using a state labeled φ and may transitions to its clauses: $\varphi \dashrightarrow^{a}{}^{S_\varphi} c_i$ for $i = 1..m$. No must transitions

are generated. See Figure D.2c and D.3. States labeled c_i represent processes resulting from translation of the individual clauses as presented above.

Observe that each satisfying assignment to formula φ has a corresponding deterministic implementation of S_φ. Also each implementation of S_φ embeds at most one satisfying assignment to φ extracted using the same rules as discussed for clauses (one per each nondeterministic choice in the initial state of the implementation). Clearly S_φ can be constructed in time polynomial in the size of φ.

We now consider construction of T_φ. First let $states_{T_\varphi} = \{true, t_\varphi, \mathbf{0}\}$. We also create the following transitions: $true \xrightarrow{a}_{T_\varphi} t_\varphi$, $true \text{-}\!\!\rightarrow^{a}_{T_\varphi} t_\varphi$, and $t_\varphi \text{-}\!\!\rightarrow^{x_i}_{T_\varphi} \mathbf{0}$ for all variables x_i of φ (See Fig. D.2d). Clearly T_φ can be constructed in time at most polynomial in size of φ.

The following lemma states the correctness of our reduction.

Lemma 1 *A 3-DNF formula φ with all satisfiable clauses is a tautology iff $[\![T_\varphi, true]\!] \subseteq [\![S_\varphi, \varphi]\!]$.*

Proof 1 *We first consider the direction right to left, i.e. assume that $[\![T_\varphi, true]\!] \subseteq [\![S_\varphi, \varphi]\!]$ and take any truth assignment ϱ to variables x_i of φ. We construct a deterministic implementation I_ϱ in the following way: $states_{I_\varrho} = \{t, \varrho, \mathbf{0}\}$, where there are two transition from t to ϱ: $t \xrightarrow{a} \varrho \wedge t \text{-}\!\!\rightarrow^{a} \varrho$ and for all x_i such that $\varrho(x_i) = true$: $\varrho \xrightarrow{x_i} \mathbf{0} \wedge \varrho \text{-}\!\!\rightarrow^{x_i} \mathbf{0}$. Due to the construction of our reduction this means that ϱ satisfies φ. Since for any assignment ϱ we can conclude that φ holds, φ is a tautology.*

Now consider the claim of the lemma from left to right. We address its contrapositive. Assume that there exists an implementation I and its state t such that $t \leq_{\mathrm{m}} true$, but $t \not\leq_{\mathrm{m}} \varphi$. We want to show that φ is not a tautology. Observe that since $t \not\leq_{\mathrm{m}} \varphi$ there must exist a state $s \in states_I$ such that $t \xrightarrow{a} s$ and for all clause states c_i of S_φ it is the case that $s \not\leq_{\mathrm{m}} c_i$. But this means that the assignment represented by s (present x_i-transitions give rise to $x_i = true$, absent to $x_i = false$) falsifies φ meaning that φ is not a tautology.□

Theorem 1 *The problem of deciding $[\![T, t]\!] \subseteq [\![S, s]\!]$ for states t and s of arbitrary modal transition systems T and S respectively is co-NP hard.*

Co-NP hardness follows from the above reduction and co-NP hardness of 3-DNF-Tautology. The same reduction can be used to show that the thorough refinement induced by weak modal refinement (Section 6) is also co-NP hard to decide. We omit that proof as the argument is rather similar to the above.

4 Syntactic Consistency and Syntactic Refinement

From now on we relax the syntactic consistency requirement presented in Section 2, and allow reasoning about systems for which $\longrightarrow \not\subseteq \text{-}\!\!\rightarrow$. We will introduce

a syntactic refinement $\subseteq_m$ with its induced notion of consistency and prove that it is (almost) precisely characterized by the syntactic consistency. These results are very simple, but we include them for three reasons. First, we cannot avoid discussing the most well known notion of consistency for modal transition systems (a notion that had never been characterized using a refinement relation). Second, we can show a refinement inducing this consistency (a refinement that had never been explicitly linked to any consistency notion). Third, we want to present all ingredients of a consistency study using a simple example: a refinement, its induced consistency, operational characterization in form of a consistency relation, and a coincidence proof. Later sections will follow exactly the same pattern.

Definition 2 (Syntactic Refinement) *For two modal transition systems S and T a syntactic refinement $\mathcal{R}$ is a partial injective function on $states_S$ into $states_T$ such that for all pairs (s,t), $t = \mathcal{R}(s)$, and all actions a it holds that*

for all $t' \in states_T$ such that $t \xrightarrow{a}_T t'$
there exists an $s' \in states_S$ such that $s \xrightarrow{a}_S s'$ and $t' = \mathcal{R}(s')$,

for all $s' \in states_S$ such that $s \dashrightarrow^a_S s'$
there exists a $t' \in states_T$ such that $t \dashrightarrow^a_T t'$ and $t' = \mathcal{R}(s')$.

A state s is said to be a syntactic refinement of a state t, written $s \subseteq_m t$, if there exists a syntactic refinement function $\mathcal{R}$ such that $t = \mathcal{R}(s)$.

Intuitively this refinement establishes that the may-transition graph of S is a subgraph of the may-transition graph of T and that the must-transition graph of T is a subgraph of the must-transition graph of S.

Definition 3 (Syntactic Consistency) *A state $s \in states_S$ is syntactically consistent iff there exists an implementation I and its state s^I such that $s^I \subseteq_m s$.*

We claim that this notion of semantic consistency (almost) coincides with the one presented in Section 2. For the sake of uniformity let us reformulate that definition using an explicit notion of consistency relation:

Definition 4 (Syntactic Consistency Relation) *Given a modal transition system S, a binary relation $\mathcal{S} \subseteq states_S \times states_S$ is a syntactic consistency relation on states of S iff for each state s if $(s,s) \in \mathcal{S}$ and each action $a \in act$ it holds that whenever $s \xrightarrow{a} s'$ for some $s' \in states_S$ then also $s \dashrightarrow^a s'$ and $(s',s') \in \mathcal{S}$.*

For a syntactic consistency relation $\mathcal{S}$ and a state $s \in states_S$ such that $(s,s) \in \mathcal{S}$, we synthesize an implementation $I_\mathcal{S}$ with a state s^I such that $s^I \subseteq_m s$. Take states of $I_\mathcal{S}$ to be consistent states of S: $states_{I_\mathcal{S}} = \{p \in states_S \mid (p,p) \in \mathcal{S}\}$ and $s^I = s$. The transition relation of $I_\mathcal{S}$ is the must transition relation of S projected on states of $I_\mathcal{S}$: $\longrightarrow^{I_\mathcal{S}} = \dashrightarrow^{I_\mathcal{S}} = \longrightarrow^S \cap (states_{I_\mathcal{S}} \times act^\tau \times states_{I_\mathcal{S}})$.

Theorem 2 (Soundness) *If there exists a syntactic consistency relation containing a state s of S then s is a syntactically consistent state in the sense of Definition 3. Moreover the implementation $I_{\mathcal{S}}$ constructed above is one of its refinements: $s^I \leq_{\mathrm{m}} s$.*

It turns out that syntactic consistency relations characterize syntactic consistency in the sense of Definition 3 in a complete manner. Given a syntactic implementation I of a modal transition system S ($I \subseteq_m S$) we can construct a syntactic consistency relation in the following way:

$$\mathcal{S}_I = \{(q, q) \in \mathit{states}_S \mid \text{exists } p \in \mathit{states}_I \wedge p \subseteq_m q\} \tag{D.2}$$

Theorem 3 (Completeness) *Let s be a state of a modal transition system S and s^I be a state of an implementation I such that $s^I \subseteq_m s$. Then there exists a syntactic consistency relation for S containing (s, s), and $\mathcal{S}_I$ is one of such.*

Since establishing consistency of models is a useful feature in modeling tools, we remark that the cost of deciding existence of syntactic implementations (via consistency relations) for a state $s \in \mathit{states}_S$ is at most (and at least) linear in the size of S. The algorithm corresponds to a traversal of the must-transition graph starting in s, and checking the consistency requirement in each state.

Syntactic consistency relations characterize syntactic consistency in the sense of [LT88] *almost* precisely. In fact the two notions coincide if all states of S are reachable from s via must transitions. Otherwise Definitions 3 and 4 allow inconsistencies in unreachable parts, which has not been taken into account in [LT88].

5 Strong Modal Refinement and Strong Consistency

In Section 2 we have recalled the notion of (strong) modal refinement. Now we introduce its induced notion of consistency and characterize it operationally.

Definition 5 (Strong Consistency) *A state s of a modal transition system S is strongly consistent iff there exists an implementation I and its state s^I such that $s^I \leq_{\mathrm{m}} s$.*

In order to give an operational characterization of strong consistency we need to lift the transition relations to sets of states. For sets $\sigma, \sigma' \subseteq \mathit{states}_S$ we write:

$$\sigma \xrightarrow{a\lfloor S\rfloor} \sigma' \quad \text{iff} \quad \exists s \in \sigma.\, \exists s' \in \sigma'.\, s \xrightarrow{a}_S s' \ , \tag{D.3}$$

$$\sigma \dashrightarrow^{a\lfloor S\rfloor} \sigma' \quad \text{iff} \quad \forall s \in \sigma.\, \exists s' \in \sigma'.\, s \dashrightarrow^{a}_S s' \ . \tag{D.4}$$

Definition 6 (Strong Consistency Relation) *Given a modal transition system S, a relation $\mathcal{B} \subseteq \mathcal{P}(states_S)$ is a strong consistency relation on $states_S$ iff for all actions $a \in act$ and all $\sigma \in \mathcal{B}$ the following condition is satisfied:*

$$\begin{array}{l} \textit{whenever } s \xrightarrow{a} {}^{S} s' \textit{ for some } s \in \sigma \textit{ and some } s' \in states_S \\ \qquad \textit{then also } \sigma \xrightarrow{a} {}^{\lfloor S \rfloor} \sigma' \textit{ and } \sigma \dashrightarrow {}^{a} {}^{\lfloor S \rfloor} \sigma' \textit{ for some } \sigma' \in \mathcal{B} \textit{ containing } s'. \end{array}$$

Elements of $\mathcal{B}$ are called consistency classes. $\mathcal{B}$ is a strong consistency relation for a state $s \in states_S$ iff it contains a consistency class σ_s such that $s \in \sigma_s$.

Given a consistency relation $\mathcal{B}$ for a state $s \in states_S$ we can synthesize an implementation $I_\mathcal{B}$ with a state $s^I \in states_{I_\mathcal{B}}$, such that $s^I \leq_\mathrm{m} s$. Take the consistency classes of $\mathcal{B}$, to be the states of $I_\mathcal{B}$: $states_{I_\mathcal{B}} = \mathcal{B}$ and s^I be the class σ_s containing s. Both transition relations of $I_\mathcal{B}$ equal the intersection of *must* and *may* transition relations of S lifted to consistency classes of $\mathcal{B}$:

$$\sigma \xrightarrow{a} {}^{I_\mathcal{B}} \sigma' \text{ and } \sigma \dashrightarrow {}^{a} {}^{I_\mathcal{B}} \sigma' \quad \text{iff} \quad \sigma \xrightarrow{a} {}^{\lfloor S \rfloor} \sigma' \text{ and } \sigma \dashrightarrow {}^{a} {}^{\lfloor S \rfloor} \sigma' \ . \tag{D.5}$$

Theorem 4 (Soundness) *If there exists a consistency relation $\mathcal{B}$ for a modal transition system S then S is strongly consistent in the sense of Definition 5. Moreover $I_\mathcal{B}$ constructed as above is one of its refinements: $s^I \leq_\mathrm{m} S$.*

Strong consistency relations characterize strong consistency in a sound and complete manner. Given a state s^I of an implementation I refining a state $s \in states_S$ ($s^I \leq_\mathrm{m} s$) we can construct a consistency relation $\mathcal{B}_I$ for S following (D.6):

$$\mathcal{B}_I = \{\sigma_p \subseteq states_S \mid p \in states_I \text{ and } \sigma_p \neq \emptyset \text{ and } \forall q \in \sigma_p . \, p \leq_\mathrm{m} q\} \tag{D.6}$$

Observe that the σ_p sets above are not necessarily maximal.

Theorem 5 (Completeness) *Let $s \in states_S$ and let I be an implementation, let $s^I \in states_I$ and $s^I \leq_\mathrm{m} s$. Then there exists a consistency relation for the state s. Also relation $\mathcal{B}_I$ defined above is one of such relations.*

Definition 6 can be interpreted operationally giving a simple exponential fixpoint algorithm: start with a singleton class containing s and apply the rule generating classes until a fixpoint is reached.

We demonstrate that the problem of deciding strong consistency is in fact NP-hard using a reduction from 3-Cnf-Sat. Let $\varphi = c_1 \wedge \ldots \wedge c_m$ be a 3-CNF formula over variables $x_1, \ldots, x_n$. Construct a modal transition system S_φ such that its state labeled c_m is consistent iff φ is satisfiable. The states of S_φ are literals of φ, a $\mathbf{0}$ state, a $\mathbf{1}$ state (a state allowing any behavior: $\mathbf{1} \dashrightarrow {}^{x_i} {}^{S_\varphi} \mathbf{1}$ for all $i = 1 \,..\, n$ and $\mathbf{1} \dashrightarrow {}^{a} {}^{S_\varphi} \mathbf{1}$), plus a polynomial number of auxiliary states. We shall use an action per each variable x_i and one auxiliary action a.

Literals in φ are translated to states using the principle shown in Figure D.2ab. A disjunction of three literals $l_1 \vee l_2 \vee l_3$ is represented by a state labeled $(l_1 \vee l_2 \vee l_3)$

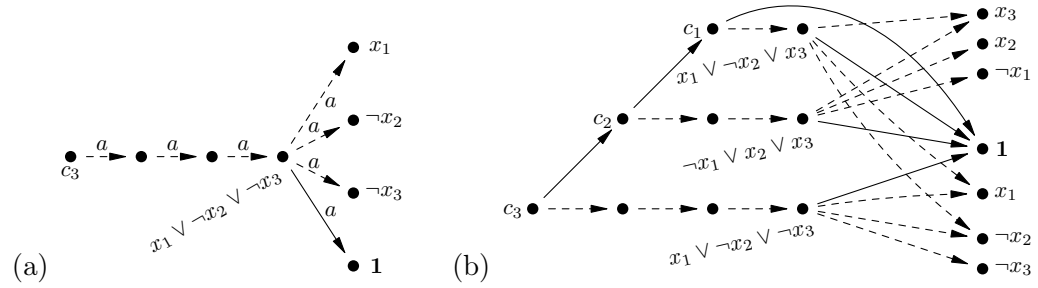


Figure D.4: Representing (a) a disjunctive clause and (b) a translation for φ.

such that $(l_1 \vee l_2 \vee l_3) \xrightarrow{a}{}^{S_\varphi} \mathbf{1}$ and $(l_1 \vee l_2 \vee l_3) \dashrightarrow^{a,S_\varphi} l_k$ for all $k = 1 \ldots 3$. Now each clause c_i is represented by a state labeled c_i followed by a sequence of exactly i may a-transitions leading to the state representing the disjunction. For regularity we assume that there is a special *true* clause c_0, that we translate to $\mathbf{1}$. Figure D.4a shows the result of translating a clause $c_3 = x_1 \vee \neg x_2 \vee \neg x_3$. Recall that states labeled with literals are actually results of translation of Figure D.2ab.

Now the top-level conjunction is translated inductively. First representations of $c_1, \ldots, c_m$ are created as above, then they are conjoined using must transitions. The ith clause is conjoined by a must transition from c_i to c_{i-1}: $c_i \xrightarrow{a}{}^{S_\varphi} c_{i-1}$. Note that we add at most a quadratic number of auxiliary states this way (and a similar number of transitions). After conjoining c_m we obtain a representation of the whole formula. Figure D.4b presents a complete translation for a formula $\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$. All unlabeled transitions should actually be labeled by a (removed to decrease clutter).

It is not hard to see that if the c_m state has an implementation then it actually has a state that satisfies the requirements of all the states representing disjunctions, and thus it induces a satisfiable assignment to φ.

6 Weak Refinement and Weak Consistency

We shall now discuss what is considered a classic form of a weak modal refinement (obtained by transforming modal refinement in the same way as bisimulation is transformed in order to obtain its weak form; to the best of our knowledge first published by Hüttel and Larsen in [HL89]). The definition uses a notion of weak transition relations that we introduce first. We shall write:

$$s \xrightarrow{a}{}_*^S s' \quad \text{iff} \quad s \left(\xrightarrow{\tau}^S\right)^* \xrightarrow{a}^S \left(\xrightarrow{\tau}^S\right)^* s' \tag{D.7}$$

$$s \dashrightarrow_*^{a,S} s' \quad \text{iff} \quad s \left(\dashrightarrow^{\tau,S}\right)^* \dashrightarrow^{a,S} \left(\dashrightarrow^{\tau,S}\right)^* s' \ , \tag{D.8}$$

where $\mathcal{R}^*$ denotes zero or more transitive applications of a binary relation $\mathcal{R}$. Finally we write $s \xrightarrow{\hat{a}}{}_*^S s'$ whenever $s \xrightarrow{a}{}_*^S s'$ and $a \neq \tau$, or whenever $s \left(\xrightarrow{\tau}^S\right)^* s'$ and $a = \tau$. Similarly for the may transition relation.

Definition 7 (Weak Modal Refinement) *Let S, T be modal transition systems. A binary relation $\mathcal{R} \subseteq states_S \times states_T$ is a weak modal refinement iff for each pair $(s,t) \in \mathcal{R}$ and each action $a \in act^\tau$ it holds that:*

for all $t' \in states_T$ such that $t \xrightarrow{a}_T t'$
there exists $s' \in states_S$ such that $s \xrightarrow{\hat{a}}_*^S s'$ and $(s',t') \in \mathcal{R}$,

for all $s' \in states_S$ such that $s \dashrightarrow^S s'$
there exists $t' \in states_T$ such that $t \dashrightarrow_*^T t'$ and $(s',t') \in \mathcal{R}$.

We say that a state $s \in states_S$ weakly refines a state $t \in states_T$, written $s \leq_{\mathrm{m}}^* t$ iff there exists a weak modal refinement containing (s,t).

Definition 8 (Weak Consistency) *A state s of a modal transition system S is weakly consistent iff there exists an implementation I and its state s^I such that $s^I \leq_{\mathrm{m}}^* s$.*

We characterize weak consistency using consistency relations as before. In order to do this we need to lift weak transition relations $\dashrightarrow_*$ and $\longrightarrow_*$ to sets of states. For two sets of states $\sigma, \sigma' \subseteq states_S$ write:

$$\sigma \xrightarrow{\hat{a}}_*^{\lfloor S \rfloor} \sigma' \quad \text{iff} \quad \exists s \in \sigma.\, \exists s' \in \sigma'.\, s \xrightarrow{\hat{a}}_*^S s' \ , \tag{D.9}$$

$$\sigma \dashrightarrow_*^{\hat{a}\,\lfloor S \rfloor} \sigma' \quad \text{iff} \quad \forall s \in \sigma.\, \exists s' \in \sigma'.\, s \dashrightarrow_*^{\hat{a}\,S} s' \ . \tag{D.10}$$

Definition 9 (Weak Consistency Relation) *Let S be a modal transition system. A relation $\mathcal{O} \subseteq \mathcal{P}(states_S)$ is a weak consistency relation on $states_S$ iff for any set $\sigma \in \mathcal{O}$, for any state $s \in \sigma$, and for any action $a \in act^\tau$ it holds that:*

whenever $s \xrightarrow{a}^S s'$ for some $s' \in states_S$
then also $\sigma \xrightarrow{\hat{a}}_*^{\lfloor S \rfloor} \sigma'$ and $\sigma \dashrightarrow_*^{\hat{a}\,\lfloor S \rfloor} \sigma'$ for some $\sigma' \in \mathcal{O}$ containing s'.

$\mathcal{O}$ is a weak consistency relation for a state $s \in states_S$ iff it contains a consistency class σ_s such that $start_s \in \sigma_s$.

As before, we claim that weak consistency relations (Definition 9) soundly characterize weak consistency (Definition 8): for a state $s \in states_S$ with a known weak consistency relation $\mathcal{O}$, one can construct a weak implementation $I_\mathcal{O}$ containing a state s^I such that $s^I \leq_{\mathrm{m}}^* s$. Take states of $I_\mathcal{O}$ to be consistency classes of $\mathcal{O}$ ($states_{I_\mathcal{O}} = \mathcal{O}$), and s^I be a class σ_s containing s. The transition relations of $I_\mathcal{O}$ are the intersection of the weak transition relations of S lifted to consistency classes of $\mathcal{O}$. For all actions $a \in act^\tau$:

$$\sigma \xrightarrow{a}^{I_\mathcal{O}} \sigma' \text{ and } \sigma \dashrightarrow^{a\,I_\mathcal{O}} \sigma' \quad \text{iff} \quad \sigma \xrightarrow{\hat{a}}_*^{\lfloor S \rfloor} \sigma' \text{ and } \sigma \dashrightarrow_*^{\hat{a}\,\lfloor S \rfloor} \sigma' \ . \tag{D.11}$$

Theorem 6 (Soundness) *Let S be a modal transition system, $s \in states_S$, and $\mathcal{O}$ be a weak consistency relation for s. Then s is weakly consistent and $s^I \in states_{I_\mathcal{O}}$ is one of its implementations: $s^I \leq_{\mathrm{m}}^* s$.*

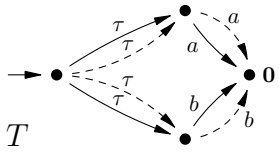


Figure D.5: All implementations of T have τ-transitions.

Consistency relations characterize weak consistency precisely. Assume that a state $s \in states_S$ is refined by a state s^I of an implementation I ($I \leq^*_{\mathrm{m}} S$). Then one can use this implementation to construct the consistency relation $\mathcal{O}_I$:

$$\mathcal{O}_I = \{\sigma_p \subseteq states_S \mid p \in states_I \text{ and } \sigma_p \neq \emptyset \text{ and } \forall q \in \sigma_p . p \leq^*_{\mathrm{m}} q\} \qquad \text{(D.12)}$$

Theorem 7 (Completeness) *Let S be a modal transition system, I be an implementation, and let $s^I \leq^*_{\mathrm{m}} s$ for some $s^I \in states_I$ and $s \in states_S$. Then there exist weak consistency relations for s, and $\mathcal{O}_I$ is one of them.*

Definition 9 can be interpreted operationally giving rise to an exponential algorithm for constructing a consistency relation and deciding weak consistency. Weak consistency collapses to strong consistency for systems without transitions labeled with τ. Consequently the problem of deciding it is at least NP-hard, by reduction from 3-Cnf-Sat presented in Section 5.

We conclude this section with a comment on synthesis of a weak implementation $I_{\mathcal{O}}$ from a consistency relation $\mathcal{O}$. The implementation synthesized by the algorithm presented above will contain internal transitions, if the specification contained them. In fact this is not always necessary—there definitely exist specifications with internal transitions that can be realized without hidden behavior. However, hidden transitions are unavoidable for some specifications. Figure D.5 shows such a specification (in fact even a syntactically consistent one).

7 May-weak Modal Refinement and Its Consistency

In [LNW07a] (Paper C) we have proposed another weakening of modal refinement, generalizing alternating simulation [AHKV98] for two players as used in interface automata [AH01]. We call it *may-weak* here, as it preserves strong behavior on must transitions, only allowing weak matching on may transitions. It has been demonstrated that may-weak modal refinement is a sound basis for assume/guarantee reasoning: it preserves absence of deadlocks on guaranteed behaviors (details in [LNW07a] (Paper C)).

Before we can define the may-weak refinement, let us define the may-weak transition relation as used in this refinement. We shall write

$$s -\!\!-\!\!\xrightarrow{a}_{\triangleleft}^{S} s' \quad \text{iff} \quad s\big(-\!\!-\!\!\xrightarrow{\tau}^{S}\big)^* s'' -\!\!-\!\!\xrightarrow{a}^{S} s' \qquad \text{(D.13)}$$

Similarly as before we write $s \text{-}\!\!-\!\!\xrightarrow{\hat{a}}_{\triangleleft}^{S} s'$ meaning $s \text{-}\!\!-\!\!\xrightarrow{a}_{\triangleleft}^{S} s'$ if $a \in act$ and $s(\text{-}\!\!-\!\!\xrightarrow{\tau}^{S})^* s'$ if $a = \tau$. We use the regular (strong) must-transition relation lifted to sets of states as in Section 5. We also lift our new may-weak transition relation:

$$\sigma \text{-}\!\!-\!\!\xrightarrow{\hat{a}}_{\triangleleft}^{\lfloor S \rfloor} \sigma' \text{ iff } \forall s \in \sigma . \exists s' \in \sigma' .\, s \text{-}\!\!-\!\!\xrightarrow{\hat{a}}_{\triangleleft}^{S} s' \ . \tag{D.14}$$

Let us now define may-weak modal refinement [LNW07a] (Paper C) using may-weak transitions:

Definition 10 (May-weak Modal Refinement) *A binary relation $\mathcal{R} \subseteq states_S \times states_T$ is a may-weak refinement between states of two modal transition systems S and T iff for each pair of states $(s,t) \in \mathcal{R}$ it holds that:*

for all $a \in act$ and for all $t' \in states_T$ such that $t \xrightarrow{a}^{T} t'$
there exists $s' \in states_S$ such that $s \xrightarrow{a}^{S} s'$ and $(s',t') \in \mathcal{R}$,

for all $a \in act^{\tau}$ and for all $s' \in states_S$ $s \text{-}\!\!-\!\!\xrightarrow{a}^{S} s'$
there exists $t' \in states_{T'}$ such that $t \text{-}\!\!-\!\!\xrightarrow{\hat{a}}_{\triangleleft}^{T} t'$ and $(s',t') \in \mathcal{R}$.

A state $s \in states_S$ may-weakly refines a state $t \in states_T$, written $s \leq_{\mathrm{m}}^{\triangleleft} t$ iff there exists a may-weak modal refinement containing (s,t).

Definition 11 (May-weak Consistency) *A state s of a modal transition system S is may-weak consistent iff there exists an implementation I and its state s^I such that $s^I \leq_{\mathrm{m}}^{\triangleleft} s$.*

Definition 12 (May-weak Consistency Relation) *Let S be a modal transition system. A relation $\mathcal{U} \subseteq \mathcal{P}(states_S)$ is a may-weak consistency relation on $states_S$ iff for any set of states $\sigma \in \mathcal{U}$, for any state $s \in \sigma$, and for any action $a \in act$ the following holds:*

whenever $s \xrightarrow{a}^{S} s'$ for some $s' \in states_S$
then also $\sigma \xrightarrow{a}^{\lfloor S \rfloor} \sigma'$ and $\sigma \text{-}\!\!-\!\!\xrightarrow{a}_{\triangleleft}^{\lfloor S \rfloor} \sigma'$ for some $\sigma' \in \mathcal{U}$ containing s'.

$\mathcal{U}$ is a may-weak consistency relation for a state $s \in states_S$ iff it contains a consistency class $\sigma_s \in \mathcal{U}$ such that $s \in \sigma_s$.

Given a consistency relation $\mathcal{U}$ for a state s of a modal transition system S, we can synthesize an implementation $I_{\mathcal{U}}$ with a state s^I refining s. The states of $I_{\mathcal{U}}$ are the consistency classes of $\mathcal{U}$: $states_{I_{\mathcal{U}}} = \mathcal{U}$ and s^I is the consistency class containing s. Transition relations of $I_{\mathcal{U}}$ equal intersection of *must* and may-weak transition relations of S lifted to consistency classes in $\mathcal{U}$ (for $a \neq \tau$):

$$\sigma \xrightarrow{a}^{I_{\mathcal{U}}} \sigma' \text{ and } \sigma \text{-}\!\!-\!\!\xrightarrow{a}^{I_{\mathcal{U}}} \sigma' \quad \text{iff} \quad \sigma \xrightarrow{a}^{\lfloor S \rfloor} \sigma' \text{ and } \sigma \text{-}\!\!-\!\!\xrightarrow{a}_{\triangleleft}^{\lfloor S \rfloor} \sigma' \ , \tag{D.15}$$

Theorem 8 (Soundness) *Let $s \in states_S$. If $\mathcal{U}$ is a may-weak consistency relation for s then s is may-weakly consistent and $s^I \in states_{I_\mathcal{U}}$ constructed as above is one of its implementations: $s^I \leq_{\mathrm{m}}^{\triangleleft} s$.*

For the completeness of characterization consider an implementation I, a state $s^I \in states_I$ such that $s^I \leq_{\mathrm{m}}^{\triangleleft} s$, where $s \in states_S$. We construct a consistency relation $\mathcal{U}_I$ for s in the following way:

$$\mathcal{U}_I = \{\sigma_p \subseteq states_S \mid p \in states_I \text{ and } \sigma_p \neq \emptyset \text{ and } \forall q \in \sigma_p.\, p \leq_{\mathrm{m}}^{\triangleleft} q\} \ . \qquad \text{(D.16)}$$

Theorem 9 (Completeness) *Let S be a modal transition system, $s \in states_S$ and let I be an implementation such that $s^I \leq_{\mathrm{m}}^{\triangleleft} s$ for some $s^I \in states_I$. Then there exist a may-weak consistency relation for s, and $\mathcal{U}_I$ is one such relation.*

Existence of a may-weak consistency relation for a given state s can be decided in exponential time, using an algorithm that is easy to extract from Definition 12. As previously this problem is also NP-hard, as may-weak consistency collapses to strong consistency for specifications without τ transitions.

A remarkable property of may-weak modal refinement, which we have not realized when writing [LNW07a] (Paper C), is that a may-weak consistent system always has implementations that contain no hidden actions ($I_\mathcal{U}$ above is actually constructed without introducing internal transitions). This is because this refinement captures a kind of (observation) determinism of required behaviors in specifications. We find this property appealing for applications again: it describes a class of specifications which allow implementations that are predictable (provided that they are deterministic). As predictability is an important property of software systems, the above decision procedure is likely to prove useful in practice.

8 Conclusion and Open Problems

We have addressed several basic questions in the theory of modal transition systems. We have shown that deciding any refinement that captures, in a precise way, the same set of concrete implementations as the standard modal refinement (or weak modal refinement) is co-NP hard. This lower bound is not tight. An upper bound of EXPTIME is easily established by casting the problem as checking satisfiability of implication between two characteristic formulas, in the modal μ-calculus. Finding a tight bound remains an open problem that we shall address shortly. We also hope to study hardness of thorough refinements induced by may-weak modal refinement and branching modal refinement [FUB06].

Furthermore we have contributed to the understanding of the relation between refinements and consistencies studying notions of consistency for modal

transition systems induced by four different refinement relations: syntactic consistency [LT88] (induced by a graph inclusion refinement), strong consistency (induced by a regular modal refinement [LT88]), weak consistency (induced by weak modal refinement [HL89]) and may-weak consistency (induced by may-weak modal refinement [LNW07a] (Paper C)). For each of these we have given a sound and complete operational characterization. The upper bound on establishing the last three of these consistencies is exponential, and they are NP-hard. Syntactic consistency can be established in linear time.

There is a range of open problems related to these results. First, it is an interesting question whether there exists a useful alternative to modal refinement that completely characterizes its own (as opposed to the currently accepted) set of implementations and that can be decided in polynomial time. The main challenge here is to argue that the set of implementations considered is interesting from a practical point of view. Alternatively, as suggested to us by Michael Huth, one can try to characterize broad classes of modal transition systems for which the currently used refinement is complete.

Finding a uniform formulation for four consistency studies as presented in this paper was a rather challenging but rewarding task. Given that they can be described so similarly one could try to take this analogy further and design a more abstract meta-consistency theory, parameterized only by a refinement.

Furthermore it is interesting to study the relation between consistency and parallel decomposition. We have done some preliminary work on that topic in [LNW07a] (Paper C), though in a rather restricted setting. We intend to generalize observational consistency of [LNW07a] (Paper C), and to understand its semantics building on the results of the present paper; ultimately employing it in a larger study of decomposition.

9 Appendix

9.1 Variations of Reduction Proof from Section 3

Weak Modal Refinement $\leq_{\mathrm{m}}^{*}$

We use the same construction as in Section 3. Just the correctness lemma is stating for a different notion of implementation, namely using weak modal refinement $\leq_{\mathrm{m}}^{*}$:

$$[\![S, s]\!]_{\leq_{\mathrm{m}}^{*}} = \{(I, i) \mid i \leq_{\mathrm{m}}^{*} s \text{ and } \longrightarrow^{I} = \dashrightarrow^{I}\} \ .$$

Lemma 2 *A 3-DNF formula φ with all satisfiable clauses is a tautology iff $[\![T_\varphi, \mathit{true}]\!]_{\leq_{\mathrm{m}}^{*}} \subseteq [\![S_\varphi, \varphi]\!]_{\leq_{\mathrm{m}}^{*}}$.*

Proof 2 *The proof in the right-to-left direction is the same as for the strong refinement. For any assignment ϱ construct a strong implementation I_ϱ that refines T_φ. From this conclude that $I_\varrho \leq_{\mathrm{m}} S_\varphi$ and consequently that ϱ satisfies φ. The same argument can be used because weak modal refinement is indeed weaker than regular (strong) modal refinement, so inclusion of all weak implementations entails inclusion of all strong implementations. Details in the main body of this paper.*

Now consider the claim of our present lemma from left to right. We address its contrapositive. Assume that there exists an implementation I and its state t such that $t \leq^*_{\mathrm{m}}$ true, but $t \not\leq^*_{\mathrm{m}} \varphi$. We want to show that φ is not a tautology.

First note that $t \xrightarrow{a}_*$ since, $t \leq^*_{\mathrm{m}}$ true. For the same reason all transitions leaving t are labeled by a. In this circumstances since $t \not\leq^*_{\mathrm{m}} \varphi$ there must exist a state $s \in states_I$ such that $t \xrightarrow{a}_* s$ and for all clause states c_i of S_φ it is the case that $s \not\leq^*_{\mathrm{m}} c_i$. Normally the refinement can also be violated by (I, t) not having some transitions required by S_φ but this cannot happen here as φ has no must transitions leaving.

We shall use s to construct an assignment ϱ such that $\varrho \not\models \varphi$. For each of the c_i states representing clauses there are only two ways to violate the refinement:

1° *Either $c_i \xrightarrow{x}$ and $s \not\xrightarrow{x}_*$,*

2° *Or $s \xrightarrow{x}_*$ and $c_i \not\xrightarrow{x}$*

This is so simple, as both systems do not have any further observable behavior after the x transitions considered, and because S_φ has no hidden transitions.

But remember that if $c_i \xrightarrow{x}$ then in our reduction this means that φ has a clause, in which x occurs as a positive literal. Similarly if $c_i \not\xrightarrow{x}$ for some c_i then it means that φ has a clause that contains x in a negative literal. Now consider an assignment ϱ such that

$$\varrho(x) = \begin{cases} 0 & \text{iff } s \not\xrightarrow{x}_* \\ 1 & \text{iff } s \xrightarrow{x}_*. \end{cases} \tag{D.17}$$

It is not hard to see that since s violates refinement with every clause state such an assignment will falsify each clause of φ. So φ is not a tautology. □

9.2 Proofs for Section 4

Proof 3 (Theorem 2) *We prove the theorem by arguing that* $s^{I_S} \subseteq_m s$. *We show that the total identity function* $\mathbf{Id}$ *on states of* I_S *is a syntactic refinement of* S *by* I_S *containing* (s^{I_S}, s). *Observe that* $\mathbf{Id}$ *is an injective function from* $states_{I_S}$ *into* $states_S$. *Let us check that* $\mathbf{Id}$ *fulfills the two requirements of Definition 2. Take a state* $p \in states_{I_S}$ *and a state* $q \in states_S$ *such that* $q = \mathbf{Id}(p)$.

1° *Let* $q \xrightarrow{a}_S q'$ *for some* $q' \in states_S$. *We need to find a state* $p' \in states_{I_S}$ *such that* $p \xrightarrow{a}_{I_S} p'$ *and* $q' = \mathbf{Id}(p')$. *It is easy to see, involving Definition 4, that* $p' = q'$ *fulfills these conditions.*

2° *Let* $p \dashrightarrow^a_{I_S} p'$ *for some* $p' \in states_{I_S}$. *We want to find a state* $q' \in states_S$ *such that* $q \dashrightarrow^a_S q'$ *and* $q' = \mathbf{Id}(p')$. *Obviously take* $q' = p'$. *Since* I_S *is an implementation we get that* $p \xrightarrow{a}_{I_S} p'$. *But this means that* $q \xrightarrow{a}_S q'$ *as the former is just a projection of the latter. Now since* $(q, q) \in \mathcal{S}$, *by Definition 4, we obtain that* $q \dashrightarrow^a_S q'$.

Since $\mathbf{Id}(s^{I_S}) = s$ *(by construction of* I_S*), we conclude that indeed* $s^{I_S} \subseteq_m s$. □

Proof 4 (Theorem 3) *We show that* $\mathcal{S}_I$ *is indeed a syntactic consistency relation in the sense of Definition 4 and that it contains* (s, s). *We begin with an observation that* $(s, s) \in \mathcal{S}_I$, *because* $s^I \subseteq_m s$. *Now consider an arbitrary pair* $(q, q) \in \mathcal{S}_I$ *and a state* $q' \in states_S$ *such that* $q \xrightarrow{a}_S q'$. *We want to show that also* $q \dashrightarrow^a_S q'$ *and* $(q', q') \in \mathcal{S}_I$. *Since* $(q, q) \in \mathcal{S}_I$ *there exists a* $p \in states_I$ *such that* $p \subseteq_m q$. *This in turn, together with* $q \xrightarrow{a}_S q'$, *implies that there exists a state* $p' \in states_I$ *such that* $p \xrightarrow{a}_I p'$ *and* $p' \subseteq_m q'$ *(so* $(q', q') \in \mathcal{S}_I$*). But* $p \xrightarrow{a}_I p'$ *is the same as* $p \dashrightarrow^a_I p'$. *Now by* $p \subseteq_m q$ *also* $q \dashrightarrow^a_S q''$ *for some* $q'' \in states_S$. *It is essential to observe that* $q'' = q'$ *because both result from applying the same refinement function to* p'. *So* $q \dashrightarrow^a_S q'$, *which finishes the proof.*□

9.3 Proofs for Section 5

Proof 5 (Theorem 4) *We show that* $s^I \in states_{I_\mathcal{B}}$ *indeed refines* $s \in states_S$, *by arguing that the following relation* $\mathcal{R}$ *is a modal refinement relation of* s *by* s^I*:*

$$\mathcal{R} = \{(\sigma, q) \in states_{I_\mathcal{B}} \times states_S \mid q \in \sigma\} \tag{D.18}$$

We consider an arbitrary pair $(\sigma, q) \in \mathcal{R}$ and prove that both requirements of Definition 1 are fulfilled.

1° *Take $q' \in states_S$ and an action a such that $q \xrightarrow{a}_S q'$. We need to find a $\sigma' \in states_{I_\mathcal{B}}$ such that $\sigma \xrightarrow{a}_{I_\mathcal{B}} \sigma'$ and $(\sigma', q') \in \mathcal{R}$. This follows right away from Definition 6. Since σ is a consistency class in $\mathcal{B}$ and $q \in \sigma$, there exists a $\sigma' \in \mathcal{B} = states_{I_\mathcal{B}}$ such that $q' \in \sigma'$, so $(\sigma', q') \in \mathcal{R}$. Also $\sigma \xrightarrow{a\lfloor S \rfloor} \sigma'$ and $\sigma \dashrightarrow^{a\lfloor S \rfloor} \sigma'$ imply that $\sigma \xrightarrow{a}_{I_\mathcal{B}} \sigma'$.*

2° *Consider $\sigma' \in states_{I_\mathcal{B}}$ and an action a such that $\sigma \dashrightarrow^{a}_{I_\mathcal{B}} \sigma'$. We need to find an $q' \in states_S$ such that $q \dashrightarrow^{a}_S q'$ and $(\sigma', q') \in \mathcal{R}$. Since $\sigma \dashrightarrow^{a}_{I_\mathcal{B}} \sigma'$, then by construction of $I_\mathcal{B}$ also $\sigma \xrightarrow{a\lfloor S \rfloor} \sigma'$ and $\sigma \dashrightarrow^{a\lfloor S \rfloor} \sigma'$. Since $q \in \sigma$ the latter, namely $\sigma \dashrightarrow^{a\lfloor S \rfloor} \sigma'$, implies that there exists a $q' \in \sigma'$ such that $q \dashrightarrow^{a}_S q'$.*

Clearly $(s^I, s) \in \mathcal{R}$, as s^I is the consistency class containing s.□

Proof 6 (Theorem 5) *We prove the theorem by showing that $\mathcal{B}_I$ is indeed a behavioral consistency relation for s in the sense of Definition 6.*

Let us first observe that because $s^I \leq_\mathrm{m} s$, the relation $\mathcal{B}_I$ is nonempty—it at least contains a consistency class containing s.

Now take an arbitrary set $\sigma_p \in \mathcal{B}_I$, an arbitrary action a and consider the requirement of Definition 6: Assume that $q \xrightarrow{a}_S q'$ for some $q \in \sigma_p$ and some $q' \in states_S$. We need to find a $\sigma_{p'} \in \mathcal{B}_I$ such that $q' \in \sigma_{p'}$ and $\sigma_p \xrightarrow{a\lfloor S \rfloor} \sigma_{p'}$ and $\sigma_p \dashrightarrow^{a\lfloor S \rfloor} \sigma_{p'}$. Since $p \leq_\mathrm{m} q$, then by definition of modal refinement it must be that there exists a p' such that $p \xrightarrow{a}_I p'$ and $p' \leq_\mathrm{m} q'$. Now let $\sigma_{p'}$ be the following set:

$$\sigma_{p'} = \{q'\} \cup \{q'' \mid \exists q \in \sigma_p.\ q \dashrightarrow^{a}_I q'' \text{ and } p' \leq_\mathrm{m} q''\} \tag{D.19}$$

Due to (D.6) *we know that there exists at least one such q'' for each $q \in \sigma_p$. This means that $\sigma_p \dashrightarrow^{a\lfloor S \rfloor} \sigma_{p'}$. Also since $q \in \sigma_p$ and $q' \in \sigma_{p'}$ (by construction) we get that $\sigma_p \xrightarrow{a\lfloor S \rfloor} \sigma_{p'}$.* □

Let us sketch briefly why our reduction from 3-CNF-SAT to strong consistency checking (Definition 5) is actually correct, or more precisely let us argue that: φ is satisfiable iff the state c_m in S_φ is strongly consistent (has an implementation).

Assume first that c_m has a strong implementation. Then it is essential to observe that there will be a path of a transitions in this implementation such that the states on that path will be refining all vertically aligned states of S_φ (using the convention of Figure D.4b). The one but last state on that a-labeled path will be a state that refines all states representing clauses. Let us call this state p. Now because all of the must transition leading to $\mathbf{1}$ in S_φ it has to be that $p \xrightarrow{a}{}^{I} p'$ for some $p' \in states_S$. Since I is an implementation then also $p \dashrightarrow^{a}{}^{I} p'$, and now by the refinement we get that p' must refine at least on of the literal states in each of the clauses, giving rise to a satisfiable assignment to φ. Note that due to the way in which literal states are constructed, it is impossible that p' refines contradictory literal states.

The argument in the opposite direction is rather simple: for a given satisfiable assignment to φ it is easy to construct a deterministic implementation of S_φ. It would have a simple path of $m+1$ transitions labeled with a leading to a state p such that $p \dashrightarrow^{x_i} \mathbf{0}$ for all x_i that are assigned *true* in the satisfying assignment.

9.4 Proofs for Section 6

The following lemma has already been known in the eighties for regular transition systems (it is mentioned by Milner in [2] for weak bisimulation equivalence). It is simple to verify that it also holds for weak modal refinement.

Lemma 3 *Consider two states (s,t) such that* $s \leq_{\mathrm{m}}^{*} t$.

for all $t' \in states_T$ *such that* $t \xrightarrow{\hat{a}}{}^{T}_{*} t'$
exists $s' \in states_S$ *such that* $s \xrightarrow{\hat{a}}{}^{S}_{*} s'$ *and* $(s', t') \in \mathcal{R}$,

for all $s' \in states_S$ *such that* $s \dashrightarrow^{\hat{a}}{}^{S}_{*} s'$
exists $t' \in states_T$ *such that* $t \dashrightarrow^{\hat{a}}{}^{T}_{*} t'$ *and* $(s', t') \in \mathcal{R}$.

Proof 7 (Theorem 6) *The proof proceeds by arguing that* $s^I \in states_{I_\mathcal{O}}$ *is indeed a refinement of* s. *This fact is witnessed by the following refinement relation:*

$$\mathcal{R} = \{(\sigma, q) \in states_{I_\mathcal{O}} \times states_S \mid q \in \sigma\} \ . \tag{D.20}$$

To check whether $\mathcal{R}$ *fulfills Definition 7 consider an arbitrary pair* $(\sigma, q) \in \mathcal{R}$.

1° *Let* $q \xrightarrow{a}{}^{S} q'$ *for some* $a \in act^\tau$ *and* $q' \in states_S$. *We need to find a* $\sigma' \in states_{I_\mathcal{O}}$ *such that* $\sigma \xrightarrow{\hat{a}}{}^{I_\mathcal{O}}_{*} \sigma'$ *and* $(\sigma', q') \in \mathcal{R}$.

Since $\mathcal{O}$ *is a consistency relation in the sense of Definition 9 there must exist a consistency class* σ' *such that* $\sigma \xrightarrow{\hat{a}}{}^{\lfloor S \rfloor}_{*} \sigma'$ *and* $\sigma \dashrightarrow^{\hat{a}}{}^{\lfloor S \rfloor}_{*} \sigma'$ *and* $q' \in \sigma'$.

[2] Citation in appendix (not included in article references) milner:1989,p. 151

The latter means that $(\sigma', q') \in \mathcal{R}$, *while the two former imply that* $\sigma \xrightarrow{a}{}^{I_{\mathcal{O}}} \sigma'$, *so also* $\sigma \xrightarrow{\hat{a}}{}^{I_{\mathcal{O}}}_{*} \sigma'$.

2° *Let* $\sigma \dashrightarrow^{a}{}^{I_{\mathcal{O}}} \sigma'$ *for some* $a \in act^{\tau}$ *and* $\sigma' \in states_{I_{\mathcal{O}}}$. *We want to find a* $q' \in states_S$ *such that* $q \dashrightarrow^{\hat{a}}_{*}{}^{S} q'$ *and* $(\sigma', q') \in \mathcal{R}$.

Since we know that $\sigma \dashrightarrow^{a}{}^{I_{\mathcal{O}}} \sigma'$ *then also* $\sigma \xrightarrow{\hat{a}}{}^{\lfloor S \rfloor}_{*} \sigma'$ *and* $\sigma \dashrightarrow^{\hat{a}}_{*}{}^{\lfloor S \rfloor} \sigma'$. *By definition* $\sigma \dashrightarrow^{\hat{a}}_{*}{}^{\lfloor S \rfloor} \sigma'$ *implies* $q \dashrightarrow^{\hat{a}}_{*}{}^{S} q'$ *for some* $q' \in \sigma'$. *Consequently* $(\sigma', q') \in \mathcal{R}$.

Finally recall that $s^I = \sigma_s \in \mathcal{O}$, *where* $s \in \sigma_s$. *Consequently* $(s^I, s) \in \mathcal{R}$, *allowing us to conclude that indeed* $s^I \leq^{*}_{\mathrm{m}} s$.□

Proof 8 (Theorem 7) *First observe that there exists at least one* $\sigma_{s^I} \in \mathcal{O}_I$ *because* $s^I \leq^{*}_{\mathrm{m}} s$ *(and it contains* s, *so* $\mathcal{O}_I$ *indeed can be a weak consistency relation for* s*).*

Then take an arbitrary $\sigma_p \in \mathcal{O}_I$, *a state* $q \in \sigma_p$, *and an action* $a \in act^{\tau}$ *such that* $q \xrightarrow{a}{}^{S} q'$ *for some* $q' \in states_S$. *We need to find a* $\sigma_{p'} \in \mathcal{O}_I$, *such that* $q' \in \sigma_{p'}$ *and* $\sigma_p \xrightarrow{\hat{a}}{}^{\lfloor S \rfloor}_{*} \sigma_{p'}$, *and* $\sigma_p \dashrightarrow^{\hat{a}}_{*}{}^{\lfloor S \rfloor} \sigma_{p'}$.

Since $p \leq^{*}_{\mathrm{m}} q$ *and* $q \xrightarrow{a}{}^{S} q'$ *we get that* $p \xrightarrow{\hat{a}}{}^{I}_{*} p'$ *and* $p' \leq^{*}_{\mathrm{m}} q'$ *for some* $p' \in states_I$. *Since* $p \xrightarrow{\hat{a}}{}^{I}_{*} p'$ *then also* $p \dashrightarrow^{\hat{a}}_{*}{}^{I} p'$ *and then, by Lemma 3, for each state* $q \in \sigma_p$ *there exists a state* $q'' \in states_S$ *such that* $q \dashrightarrow^{\hat{a}}_{*}{}^{S} q''$ *and* $p' \leq^{*}_{\mathrm{m}} q''$. *Let us now construct* $\sigma_{p'}$ *to be:*

$$\sigma_{p'} = \{q'\} \cup \{q'' \in states_S \mid \exists q \in \sigma_p.\, q \dashrightarrow^{\hat{a}}_{*}{}^{S} q'' \text{ and } p' \leq^{*}_{\mathrm{m}} q''\} \tag{D.21}$$

First, $\sigma_{p'} \in \mathcal{O}_I$ *as* p' *refines all members of* $\sigma_{p'}$. *Second, as* $q' \in \sigma_{p'}$, $\sigma_p \xrightarrow{a}{}^{\lfloor S \rfloor} \sigma_{i'}$, *which is witnessed by* $q \xrightarrow{a}{}^{S} q'$, *which is stronger than* $\sigma_p \xrightarrow{\hat{a}}{}^{\lfloor S \rfloor}_{*} \sigma_{p'}$. *Finally since all* q''*'s in* σ_p *have weak* a*-successors in* $\sigma_{p'}$, *get that* $\sigma_p \dashrightarrow^{\hat{a}}_{*}{}^{\lfloor S \rfloor} \sigma_{p'}$, *by definition of the weak lifted transition relation.*□

9.5 Proofs for Section 7

Proof 9 (Theorem 8) *We prove the theorem by arguing that the following relation $\mathcal{R}$ indeed is a may-weak modal refinement of s by s^I.*

$$\mathcal{R} = \{(\sigma, q) \in states_{I_\mathcal{U}} \times states_S \mid q \in \sigma\} \tag{D.22}$$

We consider an arbitrary pair $(\sigma, q) \in \mathcal{R}$ and check that both requirements of Definition 10 are fulfilled.

1° *Take $q' \in states_S$ and an action $a \in act$ such that $q \xrightarrow{a}^S q'$. We need to find a $\sigma' \in states_{I_\mathcal{U}}$ such that $\sigma \xrightarrow{a}^{I_\mathcal{U}} \sigma'$ and $(\sigma', q') \in \mathcal{R}$.*

Since $q \xrightarrow{a}^S q'$ the definition of may-weak consistency relation (Definition 12) gives us that $\sigma \xrightarrow{a\lfloor S \rfloor} \sigma'$ and $\sigma \dashrightarrow_\triangleleft^{a\lfloor S \rfloor} \sigma'$ for some $\sigma' \in \mathcal{U}$ such that $q' \in \sigma'$. Which exactly gives us $\sigma \xrightarrow{a}^{I_\mathcal{U}} \sigma'$ and $\sigma' \in \mathcal{U}$ gives us $(\sigma', q') \in \mathcal{R}$.

2° *Consider $\sigma' \in states_{I_\mathcal{U}}$ and an action $a \in act^\tau$ such that $\sigma \dashrightarrow^{a}_{\triangleleft}{}^{I_\mathcal{U}} \sigma'$. We need to find a state $q' \in states_S$ such that $q \dashrightarrow_\triangleleft^{\hat{a}}{}^S q'$ and $(\sigma', q') \in \mathcal{R}$.*

Since $\sigma \dashrightarrow^{a}_{\triangleleft}{}^{I_\mathcal{U}} \sigma'$ then by construction of $I_\mathcal{U}$ also $\sigma \xrightarrow{a\lfloor S \rfloor} \sigma'$ and $\sigma \dashrightarrow_\triangleleft^{\hat{a}}{}^S \sigma'$. Since $q \in \sigma$, the latter, namely $\sigma \dashrightarrow_\triangleleft^{\hat{a}}{}^S \sigma'$, implies that there exists $q' \in \sigma'$ such that $q \dashrightarrow_\triangleleft^{\hat{a}}{}^S q'$ and $(\sigma', q') \in \mathcal{R}$.

Finally recall that $s^I \in \mathcal{U}$ and $s \in s^I$, which allows us to conclude that indeed $s^I \leq_{\mathrm{m}}^\triangleleft s$ as witnessed by $\mathcal{R}$.□

Proof 10 (of Theorem 9) *We prove the theorem by showing that $\mathcal{U}_I$ is indeed a consistency relation in the sense of Definition 12.*

Let us begin with a simple observation that because $s^I \leq_{\mathrm{m}}^\triangleleft start_s$ the relation $\mathcal{U}_I$ is non-empty: it at least contains the set $\sigma_s = \{s\}$. Now take an arbitrary set $\sigma_p \in \mathcal{U}_I$ and an arbitrary action $a \in act$ and consider the requirement of Definition 12. Assume that $q \xrightarrow{a}^S q'$ for some $q \in \sigma_p$ and some $q' \in states_S$. We need to find a $\sigma_{p'} \in \mathcal{U}_I$ such that $q' \in \sigma_{p'}$ and $\sigma \xrightarrow{a\lfloor S \rfloor} \sigma_{p'}$ and $\sigma \dashrightarrow_\triangleleft^{a\lfloor S \rfloor} \sigma_{p'}$.

Since $p \leq^{\triangleleft}_{\mathrm{m}} q$ *then by Definition 10 and our assumption* $(q \xrightarrow{a}_{S} q')$ *it must be that there exists a* $p' \in states_I$ *such that* $p \xrightarrow{a}_{Iu} p'$ *and* $p' \leq^{\triangleleft}_{\mathrm{m}} q'$. *Now let* $\sigma_{p'}$ *be the following set:*

$$\sigma_{p'} = \underline{\{q'\}} \cup \{q'' \in states_S \mid \exists q \in \sigma_p.\ q \dashrightarrow^{a}_{\triangleleft} q'' \text{ and } p' \leq^{\triangleleft}_{\mathrm{m}} q''\} \tag{D.23}$$

Due to (D.16) *we know that there exists at least one such* q'' *for each* $q \in \sigma_p$, *which means that* $\underline{\sigma_p \dashrightarrow^{a}_{\triangleleft}{}^{\lfloor S \rfloor} \sigma_{p'}}$. *Also since* $q \in \sigma_p$ *and* $q' \in \sigma_{p'}$ *(by construction) we get that* $\underline{\sigma_p \xrightarrow{a}{}^{\lfloor S \rfloor} \sigma_{p'}}$.□

Paper E

Complexity of Decision Problems for Mixed and Modal Specifications

Adam Antonik, Michael Huth
Department of Computing,
Imperial College London, United Kingdom

Kim G. Larsen, Ulrik Nyman
Department of Computer Science,
Aalborg University, Denmark

Andrzej Wąsowski
Computational Logic and Algorithms Group,
IT University of Copenhagen, Denmark

Abstract[1]

We consider decision problems for modal and mixed transition systems used as specifications: the *common implementation* problem (whether a set of specifications has a common implementation), the *consistency* problem (whether a single specification has an implementation), and the *thorough refinement* problem (whether all implementations of one specification are also implementations of another one). *Common implementation* and *thorough refinement* are shown to be PSPACE-hard for modal, and so also for mixed, specifications. *Consistency* is PSPACE-hard for mixed, while trivial for modal specifications. We also supply upper bounds suggesting strong links between these problems.
Keywords: Modal Transition Systems, Common Implementation, Thorough Refinement, Consistency, Modal Specifications, Mixed Specifications

[1]Partially supported by the UK EPSRC projects *Efficient Specification Pattern Library for Model Validation (EP/D50595X/1)* and *Complete and Efficient Checks for Branching-Time Abstractions (EP/E028985/1)*

1 Introduction

Bisimulation equivalence [Mil89, Par81] is widely accepted as a correctness criterion for realizations of abstract specifications. Bisimulation is, however, a rather strong relation that severely, and often unnecessarily, limits the choices of designers in how specifications should be realized. At the same time, the main alternative, bisimulation's sister preorder *simulation* [Mil89], is often too weak to use in this context as it only limits faulty behaviours, without enforcing any correct ones.

In order to address these shortcomings, Larsen and Thomsen [LT88] have proposed *modal transition systems* and the accompanying *modal refinement*, in this paper referred to simply as refinement. Modal transition systems feature required and allowed transitions able to simultaneously describe an under- and over-approximation of behavior within a single specification. Modal refinement generalizes both simulation and bisimulation, letting the specifier choose the required level of strictness in the spectrum between the two. In [Lar89] Larsen argued that any sufficiently expressive specification language necessarily must accommodate inconsistent specifications, akin to inconsistent logical formulæ, and thus lifted the consistency requirement. The same type of systems were independently reintroduced by Dams as *mixed transition systems* [Dam96, DGG97].

Here we establish complexities of several decision procedures for this family of specification languages, addressing several long outstanding open problems:

CI Deciding whether $k > 1$ modal transition systems have a *common implementation* is PSPACE-hard in the sum of the sizes of these k systems.

C Deciding whether a mixed transition system is *consistent*, i.e. whether it has an implementation, is PSPACE-hard in the size of that system.

TR Deciding whether one modal transition system *thoroughly refines* another modal transition system is PSPACE-hard in the size of these systems.

We show quite strong links between these problems. In particular we efficiently reduce problems of type CI to problems of type C, and problems of type C to problems of type TR—though *mixed*, not necessarily modal, transition systems are the targets of that latter reduction. All three problems C, CI, and TR are shown to be in EXPTIME.

We begin with discussing the related work in Section 2 and introducing the basic concepts in Section 3. The hardness results and the aforementioned problem reductions for common implementation, consistency, and thorough refinement are the subject of Sections 4, 5, and 6 respectively. A general discussion, including the provision of upper bounds, is given in Section 7. We conclude in Section 8.

2 Related Work

Our terminology differs from that used in [LNW07c] (Paper D): what we call "modal transition systems" and "mixed transition systems" are called respectively "syntactically consistent modal transition systems" and "modal transition systems" therein.

In [HH04] a superpolynomial algorithm was given for deciding CI for $k > 1$ modal specifications. The algorithm is exponential in k, but polynomial if k is fixed. In particular, it computes a common implementation if there is one. These upper bounds also follow easily from the polynomial algorithm for consistency checking of a conjunction of disjunctive modal transition systems, as studied in [LX90].

Larsen et al. [LNW07c] (Paper D) show that TR is coNP-hard, while C is NP-hard. We strengthen both of these bounds here. They also hint at exponential upper bounds for both problems, without arguing how these can be achieved. We elaborate on how to attain these bounds, by giving precise reductions in Section 7.

Hussain and Huth [HH06] present an example of two modal specifications that have a common implementation but no greatest common implementation.

Fischbein et al. [FUB06] use modal specifications for behavioral conformance checking of products with specifications of product families. They propose a new thorough refinement whose implementations are defined through a refinement notion that generalizes branching bisimulation. The thorough refinement obtained in this manner is finer than weak refinement, and argued to be more suitable for conformance checking. In the light of the present work it is very likely that this refinement can be shown to be PSPACE-hard in the size of the specifications.

3 Background

Let us begin with defining the basic objects of interest in our study [Lar89, Dam96, CGL94]:

Definition 1 *For an action alphabet Σ, a* mixed transition system *M is a triple $(S, R^{\Box}, R^{\Diamond})$, where S is a set of states and $R^{\Box}, R^{\Diamond} \subseteq S \times \Sigma \times S$ are must- and may-transitions relations respectively. A* modal transition system *is a mixed transition system satisfying $R^{\Box} \subseteq R^{\Diamond}$; all its must-transitions are also may-transitions. A* pointed *mixed (respectively modal) transition system (M, s) is a mixed (modal) transition system M with a designated initial state $s \in S$. The size $|M|$ of a mixed (modal) transition system M is defined as $|S| + ||R^{\Box} \cup R^{\Diamond}$. All transition systems considered here are finite, i.e. Σ and S are always finite sets.*

Throughout this paper we refer to pointed modal (mixed) transition systems as modal (mixed) *specifications*. Throughout figures, solid arrows denote $R^{\Box}$-transitions, dashed arrows denote $R^{\Diamond}$-transitions. Arrows without labels have

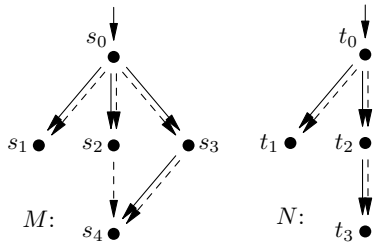


Figure E.1: Specifications (M, s_0), (N, t_0) with $I(M, s_0) = I(N, t_0)$ (so $I(M, s_0) \subseteq I(N, t_0)$), but not $(N, t_0) \prec (M, s_0)$

an implicit $\star$-label, where $\star \in \Sigma$ is an action with context-dependent meaning. Two examples of modal specifications are depicted in Fig. E.1, while a mixed specification that is not a modal specification can be seen in Fig. E.5.

Modal refinement [Lar89, Dam96, CGL94] is a refinement relationship for mixed specifications that allows verifying that one such specification is more abstract than another. It generalizes bisimulation [Par81] to underspecified models:

Definition 2 *A mixed specifications* $(N, t_0) = ((S_N, R_N^\Box, R_N^\Diamond), t_0)$ refines *another mixed specification* $(M, s_0) = ((S_M, R_M^\Box, R_M^\Diamond), s_0)$ *over the same alphabet, written* $(M, s_0) \prec (N, t_0)$*, iff there is a relation* $Q \subseteq S_M \times S_N$ *containing* (s_0, t_0) *and whenever* $(s, t) \in Q$ *then*

1° *for all* $(s, a, s') \in R_M^\Box$ *there exists some* $(t, a, t') \in R_N^\Box$ *with* $(s', t') \in Q$.

2° *for all* $(t, a, t') \in R_N^\Diamond$ *there exists some* $(s, a, s') \in R_M^\Diamond$ *with* $(s', t') \in Q$.

Deciding whether one finite-state mixed specification refines another one is in P.

Labeled transition systems over an alphabet Σ are pairs (S, R) where S is a set of states and $R \subseteq S \times \Sigma \times S$ is a transition relation. We identify labeled transition systems (S, R) with modal transition systems (S, R, R). The set of implementations $I(M, s)$ of a mixed specification (M, s) are all pointed labeled transition systems (T, t) refining (M, s). Note that $I(M, s)$ may be empty in general, but is guaranteed to be non-empty if M is a modal transition system.

Example. (Due to Harald Fecher) Figure E.1 shows modal specifications (M, s_0) and (N, t_0) over alphabet $\{\star\}$. Relation $Q = \{(s_0, t_0), (s_1, t_1), (s_2, t_2), (s_3, t_2), (s_4, t_3)\}$ witnesses that (N, t_0) refines (M, s_0), but (M, s_0) does not refine (N, t_0).

As in [LNW07c] (Paper D) we define the *thorough refinement* $(M, s) \prec_{th} (N, t)$ to be the predicate $I(N, t) \subseteq I(M, s)$. Transitivity of refinement ensures that refinement soundly characterizes thorough refinement: $(M, s) \prec (N, t)$ implies $(M, s) \prec_{th} (N, t)$. But the converse does not hold: completeness of refinement for thorough refinement is known to be false [Hüt88, Xin92, SF07]; Figure E.1 provides a counterexample.

We shall now formally define the problems that we study, and briefly discuss their significance.

Common implementation (CI): given $k > 1$ mixed specifications (M_i, s_i), is the set $\bigcap_{i=1}^{k} I(M_i, s_i)$ non-empty? For example, (M_1, s_1) could be our system model and all other (M_i, s_i) could be definitions of faulty behavior (respectively features). Common implementations are then possible implementations of our model that can exhibit all $k-1$ faults (features).

Consistency (C): Is $I(M, s)$ non-empty for a mixed specification (M, s)? Specification formalisms need the ability to express inconsistencies so that conflicts in systems or their design are detectable. Equally, inconsistent specifications may well result from the composition of consistent specifications.

Thorough refinement (TR): Does a mixed specification (N, t) thoroughly refine a mixed specification (M, s), i.e., do we have $I(N, t) \subseteq I(M, s)$? As refinement is only sound but not complete for thorough refinement, the question arises of whether thorough refinement has an efficient, e.g. co-inductive, definition that can be integrated in refinement tools.

We assume that specifications are finite-state, given their abstract nature. But implementations may (have to) be infinite-state as we otherwise cannot express important features, e.g. unbounded ranges of data types. For the three decision problems studied in this paper, it turns out that they won't change if we restrict implementations to finite-state ones. For example, a mixed specification (M, s) is consistent in the infinite sense iff its characteristic modal mu-calculus formula $\Psi_{(M,s)}$ [Hut05a] is satisfiable. Appealing to the small model theorem for that logic, $\Psi_{(M,s)}$ is satisfiable iff it is satisfiable over finite-state implementations. We can reason in a similar manner about common implementation, through the formula $\bigwedge_i \Psi_{(M_i,s_i)}$. Finally, $(M, s) \prec_{th} (N, t)$ is false iff $\Psi_{(N,t)} \wedge \neg\Psi_{(M,s)}$ is satisfiable. This justifies that we consider only finite-state specifications and implementations.

Throughout this paper we work with Karp reductions, many-one reductions computable by deterministic Turing machines in polynomial time. This choice is justified since we reduce problems that are PSPACE-complete.

4 Common Implementation

We show that the CI problem is PSPACE-hard for modal specifications, which then automatically renders the same hardness result for mixed specifications.

Theorem 1 *Let* $\{(M_i, s_i) \mid 1 \leq i \leq k\}$ *with* $k > 1$ *be a finite family of modal specifications over the same action alphabet* Σ. *Deciding emptiness of the set* $\bigcap_{i=1}^{k} I(M_i, s_i)$ *is PSPACE-hard in* $\sum_{i=1}^{k} |M_i|$.

We argue for this by reduction from the Generalized Geography game [GJ79, JL91].

Definition 3 *A rooted, directed graph is a structure $G = (V, E, v_0)$, where V is a finite set of vertices, $E \subseteq V \times V$ is a set of edges and $v_0 \in V$ is the root. For an edge $e = (u, v) \in E$ we write $\mathbf{tgt}\, e$ for v and $\mathbf{src}\, e$ for u, and we define $Follow(e) := \{f \in E \mid \mathbf{tgt}\, e = \mathbf{src}\, f\}$ and $Init := \{e \in E \mid \mathbf{src}\, e = v_0\}$.*

For $G = (V, E, v_0)$ the two-player *Generalized Geography* game on G is played according to the following rules:

> "The two players alternate choosing a new edge from E. The first edge chosen (by player 1) must have its source at v_0 and each subsequently chosen edge must have its source at the vertex that was the target of the previous edge and must not have been previously chosen in the game. The first player unable to choose such a new edge loses." [GJ79, p. 254]

The generalized geography problem (GenGeo) is whether given a rooted directed graph G does there exist a winning strategy for player 1 in the *Generalized Geography* game played on G? GenGeo is PSPACE-complete [GJ79].

Proof 1 (of Theorem 1) *We reduce* GenGeo *to checking CI of k modal specifications $\{(M_i, s_i)\}$, where both k and each $|M_i|$ are at most polynomial in the size of G. The reduction should be such that a common implementation of all (M_i, s_i), if it exists, will explicitly give the winning strategy for Player 1.*

We will create a set of modal specifications for each kind of conditions imposed by the game. All specifications will share an alphabet $\Sigma = E \cup \{\star\}$, where $\star$ is a fresh name such that $\star \notin E$. Choosing an edge in the game corresponds to taking a transition in these specifications.

Let us begin with modal specifications (P_1, s_1) and (P_2, s_2) presented in Figure E.2, which ensure that Player 1 can always continue – a necessary condition for obtaining a winning strategy. Transitions with labels $X \subseteq \Sigma$ denote sets of transitions, one for each $e \in X$. We keep track of whose turn it is in the game by distinguishing Player 1 states from Player 2 states, labeling states with Player numbers for the sake of clarity. Observe that both P_1 and P_2 oscillate between Player 1 and Player 2 decisions. Each Player 2 move is modeled directly by a single transition, while a Player 1 move is modeled by exactly two transitions; a $\star$-transition followed by a regular edge transition. As will be seen later, disjunctive choices will only occur in Player 1 mode, so $\star$-transitions used to encode disjunctions are there only for Player 1 states. Specification P_1 limits choices of Player 1 to a disjunction of all legal actions, while P_2 enforces that at least one of these choices is indeed taken.

Let us continue with the remaining GenGeo *game rules. We can enforce that an edge e is played at most once using a modal specification (M_e, s_e) shown in the left part of Figure E.3. This specification models a flag that disallows any further e-transitions once e has been used. Similarly, for each edge e create a*

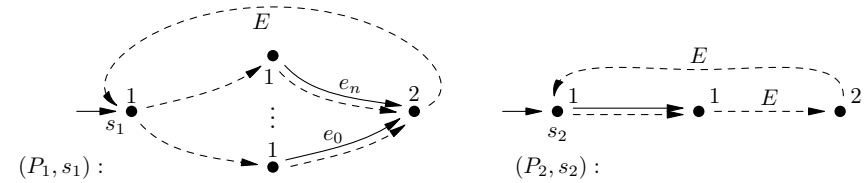


Figure E.2: Modal specifications (P_1, s_1) and (P_2, s_2) together ensuring that Player 1 can always continue playing. Assume $E = \{e_0, \ldots e_n\}$

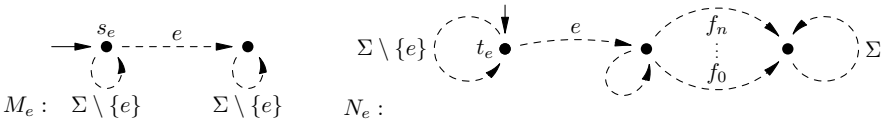


Figure E.3: Specifications M_e, N_e instantiated for each $e \in E$ and $\{f_0, \ldots, f_n\} = Follow(e)$

modal specification (N_e, t_e), as shown in the right part of Figure E.3, to constrain the moves following an e move to edges directly following it. N_e has a $\star$-labeled loop on its middle state to account for both Player 1 and Player 2 moves. Recall that if e was played by Player 2, then in our encoding it will be first followed by a $\star$ before Player 1 plays any subsequent edge. The requirement that Player 1 should choose one of the transitions leaving the root as the first move is enforced by (P_0, s_0) as shown in the left part of Figure E.4.

We are left with the last and the most complex game rule, namely that whenever Player 1 makes a choice then Player 2 has to be able to respond with any so far unused edge f following that choice. Our implementation, which directly represents the strategy, should thus have all transitions representing possible choices in such a state. We model this by creating a specification (M_{ef}, s_{ef}) for every pair of edges e and f such that $f \in Follow(e) \setminus \{e\}$. The idea is that each modal transition system M_{ef} enforces an f transition after an e transition has been chosen by Player 1, unless f has already been used (either by Player 1 or Player 2), or e has been used by Player 2. See the right part of Figure E.4.

The answer to GenGeo(V, E, v_0) *is yes iff the answer to CI is yes for*

$$\left(\bigcup_{i=0..2} \{(P_i, s_i)\} \right) \cup \bigcup_{e \in E} \left(\{(M_e, s_e), (N_e, t_e)\} \cup \bigcup_{f \in Follow(e) \setminus \{e\}} \{(M_{ef}, s_{ef})\} \right) \quad . \qquad \text{(E.1)}$$

The size of each of these $O(|E|^2)$ specifications is $O(|E|)$.□

Corollary 2 *The common implementation problem for $k > 1$ mixed specifications is PSPACE-hard in the size of these specifications.*

Proof 2 *This follows from Theorem 1 and the fact that the set of mixed specifications is a superset of the set of modal specifications.*□

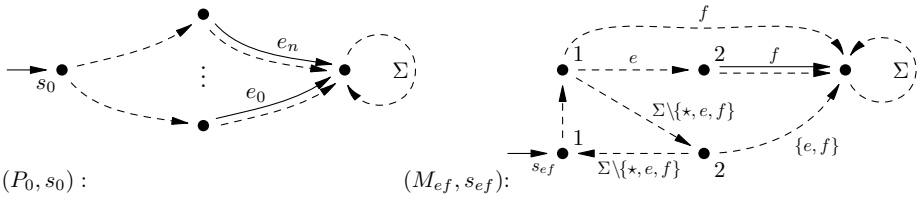


Figure E.4: Specifications (P_0, s_0) and (M_{ef}, s_{ef}) assuming that $\mathit{Init} = \{e_0, \ldots, e_n\}$

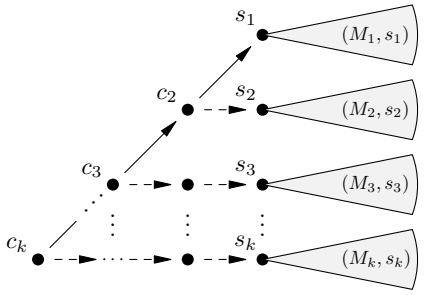


Figure E.5: Conjunction of k mixed specifications into one mixed specification

5 Consistency

Let us now show that consistency of a single mixed specification is PSPACE-hard in its size. We achieve this by appealing to Theorem 1, and reducing CI for several modal specifications to the C for a single mixed specification.

Theorem 3 *Consistency of a mixed specification is PSPACE-hard.*

Proof 3 *By Theorem 1, it suffices to show how $k > 1$ mixed specifications (M_i, s_i) can be conjoined into one mixed specification (M, c_k) with $|\, M \,|$ being polynomial in $\sum_i |\, M_i \,|$ such that (M, c_k) has an implementation iff all (M_i, s_i) have a common implementation.*

Figure E.5 illustrates the construction, which originates in [LNW07c] (Paper D), by showing a conjunction of states s_1, s_2, s_3 up to s_k. In order to conjoin two states s_1 and s_2, two new $\star$-transitions are added from a fresh state c_2 to each of s_1, s_2. One of the $\star$-transitions is a may $\star$-transition and the other is a must $\star$-transition. Only two states can be conjoined directly in this way, but the process can be iterated as many times as needed, as seen in the figure, by adding a corresponding number of $\star$-transitions to the newly conjoined systems. Observe that the resulting specification is properly mixed (not modal). Its size is linear in $\sum_i |\, M_i \,|$ and quadratic in k, which itself is $O(\sum_i |\, M_i \,|)$.

If the specifications that are being conjoined have a common implementation, then the new specification will also have an implementation which is the same implementation prefixed with a sequence of $k - 1$ $\star$-transitions. Conversely if

the new mixed specification has an implementation, then this implementation will contain at least a sequence of $k-1$ $\star$-transitions, followed by an implementation that must individually satisfy all the systems that have been conjoined.

6 Thorough Refinement

We show PSPACE-hardness of TR for mixed specifications by appeal to Theorem 3 and a reduction of consistency checks to thorough refinement checks.

Theorem 4 *Thorough refinement of mixed specifications is PSPACE-hard in the size of these specifications.*

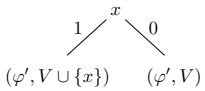
Proof 4 *By Theorem 3 deciding C for a mixed specification is PSPACE-hard. Therefore it suffices to reduce C to TR. Let (M,s) be a mixed specification over Σ. Consider a modal specification (N,t) over $\Sigma \cup \{\star\}$ with $N = (\{t\},\{\},\{\})$, which only has a single state and no transitions. From (M,s) construct the mixed specification (M',s') over $\Sigma \cup \{\star\}$ by prefixing s with a new state s' and a single transition $(s',\star,s) \in R^{\Diamond}_{M'} \setminus R^{\Box}_{M'}$. Then (M',s') is a mixed specification that has (N,t) as an implementation, where $Q = \{(s',t)\}$ is the witnessing refinement relation. We show that (M,s) is consistent iff not $(N,t) \prec_{th} (M',s')$.*

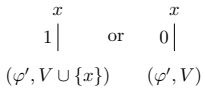
1° *If (M,s) is consistent, then it has an implementation (L,l), from which we get an implementation (L',l') of (M',s') by creating a new state l' with a transition $(l',\star,l)$. But then (M',s') has an implementation that is not allowed by (N,t) and so $I(M',s') \not\subseteq I(N,t)$.*

2° *Conversely, if $I(M',s') \not\subseteq I(N,t)$ then there exists an implementation (L,l') of (M',s'), which is not an implementation of (N,t) – and so (L,l') has a transition $(l',\star,l)$. Moreover (L,l) refines (M,s) since (L,l') refines (M',s') and s is the unique successor of s' in M'. Thus (M,s) is consistent.*

Remark: Observe that the first argument above would also work for refinement instead of thorough refinement. However we would not be able to get the second implication for refinement, due to its incompleteness.□

Let us now strengthen Theorem 4 to the subclass of *modal* specifications, by a polynomial reduction from the PSPACE-complete decision problem QUANTIFIED 3SAT [GJ79, pp. 171-2] of computing the truth value of closed quantified Boolean formulæ in 3CNF. These formulæ are of the form $Qx_1 \ldots Qx_n.\, \chi$, where each Q is $\exists$ or $\forall$ and χ is a propositional formula over $x_1,\ldots,x_n$ in 3CNF. We refer to them as QCNF formulæ in here. We can assume without loss of generality that our formulæ do not contain any clauses with duplicate literals, nor vacuously true clauses. We use $\forall x\, \exists y\, (\neg x \vee y) \wedge (\neg y \vee x)$ as a running example.

$(\forall x\varphi', V) \rightsquigarrow$ 

$(\exists x\varphi', V) \rightsquigarrow$ 

$(\chi', V) \rightsquigarrow \quad V$

Figure E.6: Semantics of QCNF as a non-deterministic rewrite system

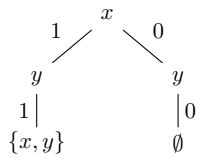


Figure E.7: Valuation tree witnessing the truth of $\forall x\,\exists y\,(\neg x \vee y) \wedge (\neg y \vee x)$

We present the semantics of QCNF formulæ in a style that will facilitate our proof. Each formula φ can be rewritten into a set of *valuation trees*. The non-deterministic rewrite system for this is depicted in Figure E.6. Universal quantification rewrites into branching, existential quantification into a choice, and the 3CNF kernel χ into the set of variables selected to be true on the path from the tree root to that kernel node. The terminals of this rewrite system for term $(\varphi, \emptyset)$ are valuation trees of φ. One such valuation tree for the formula $\forall x\,\exists y\,(\neg x \vee y) \wedge (\neg y \vee x)$ can be seen in Figure E.7. Each leaf of a valuation tree T contains all those x_i that are true in the respective model for the propositional kernel formula χ. We define $T \models \varphi$ to mean that all models of leaves of T satisfy the kernel χ of φ. Finally, φ is defined to be true iff there is a valuation tree T for φ such that $T \models \varphi$. For example, $T \models \varphi$ for the valuation tree T in Fig. E.7, as the CNF kernel $(\neg x \vee y) \wedge (\neg y \vee x)$ is true in both models $\{x, y\}$ (x and y are true) and $\emptyset$ (x and y are false). Thus, φ is true.

In Figure E.8 we present a second non-deterministic rewrite system whose terminals are *potential valuation trees*. In this new system there is no path context, existential quantification has two more rewrite rules, and the CNF kernel may rewrite into any subset of its variables. The terminals of this rewrite system are *potential* valuation trees of φ. By construction, every valuation tree is a potential valuation tree. A potential valuation tree that is not a valuation tree is called a *flawed valuation tree*. Figure E.9 shows a valuation tree for our running example with three kinds of flaws: the leftmost y node has no successor, the rightmost y node has two successors, and the leaf set $\{x, y\}$ is inconsistent with the 0 label for x on its path.

Our reduction constructs for any φ of QCNF two modal specifications (N_φ, t_φ) and (M_φ, s_φ) such that

$$I(N_\varphi, t_\varphi) \subseteq I(M_\varphi, s_\varphi) \qquad \text{iff} \qquad \varphi \text{ is false.} \tag{E.2}$$

The intuition behind the construction is that (N_φ, t_φ) models potential valuation trees and (M_φ, s_φ) models flawed, and only flawed, valuation trees of φ.

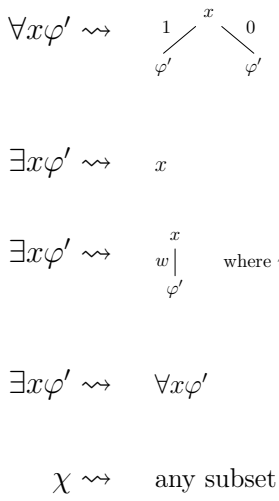


Figure E.8: Non-deterministic rewrite system for QCNF deriving *potential* valuation trees

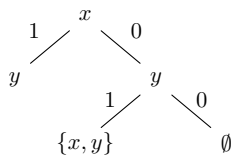


Figure E.9: Flawed valuation tree for formula $\forall x\, \exists y\, (\neg x \vee y) \wedge (\neg y \vee x)$

More precisely, these modal specifications are such that any valuation tree T with $T \models \varphi$ can be transformed into an implementation of (N_φ, t_φ) that is not an implementation of (M_φ, s_φ) and, conversely, that any element of $I(N_\varphi, t_\varphi) \setminus I(M_\varphi, s_\varphi)$ can be transformed into such a valuation tree T with $T \models \varphi$.
Both models are defined over the following alphabet

$$\Sigma_\varphi = \{\star\} \cup \{vx_i, v\neg x_i \mid 1 \leq i \leq n\} \tag{E.3}$$

where $x_1, \ldots, x_n$ is the set of variables of φ.[2] Specification (N_φ, t_φ) is defined by structural induction on φ according to the rules presented in Figure E.11.

The initial state t_φ has a must $\star$-transition to the continuation of the compilation of N_φ. Each quantifier Qx_i gets translated into a diamond shaped model of $\star$-transitions, where the upper half consists of must and may transitions for quantifiers $\forall$ and $\exists$ (respectively). The corners of diamonds have "spikes", transitions labeled with a "truth value" v_{x_i} or $v_{\neg x_i}$, for quantifier variable x_i, to a dead-end state. After all quantifiers have been compiled in this manner, conjunction is compiled as a fork of two must $\star$-transitions, disjunction as a fork of two may $\star$-transitions, and literals compiled as spikes of truth values. See the result of this compilation for our running example in Figure E.12.

Refinement, as defined for modal specifications, does not guarantee that a fork of may $\star$-transitions (present in the compilation of $\exists x_i$ and $\vee$) will implement at least one of these may $\star$-transitions. Also, an implementation may be inconsistent as to its choice of truth values v_{x_i} or $v_{\neg x_i}$. Each path through a

[2] A stronger, albeit more complicated, reduction is possible to TR of specifications over a singleton alphabet. We show the simpler variant here for the sake of clarity.

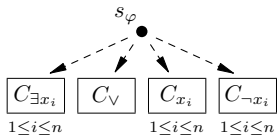


Figure E.10: Structure of modal specification (M_φ, s_φ): $\star$-transitions lead from s_φ to components that detect possible flaws in potential valuation trees of φ

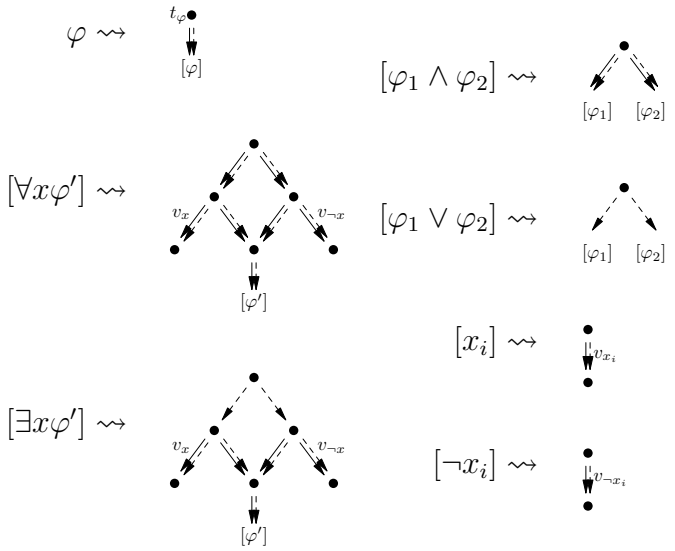


Figure E.11: Deterministic rules rewriting a QCNF formula φ into a specification (N_φ, t_φ)

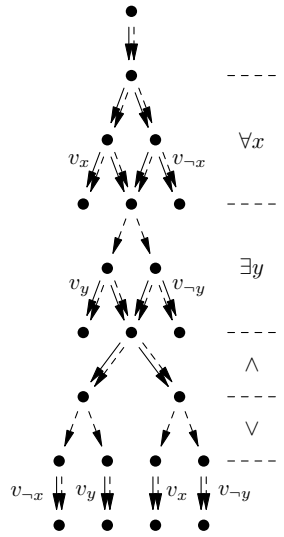


Figure E.12: Modal specification (N_φ, t_φ) for $\varphi = \forall x \, \exists y \, (\neg x \vee y) \wedge (\neg y \vee x)$

sequence of diamonds corresponds to a choice of such truth values, recorded in the respective spike transition. When such a path reaches the compilation of a propositional literal, that literal may well be inconsistent with the spike for that literal encountered en route. In total, these are then the static criteria for corresponding to a flawed valuation tree, and hence drive the construction of specification (M_φ, s_φ), whose architecture is depicted in Fig. E.10. Initial state s_φ has may $\star$-transitions to modal specifications, components that each encode a potential flaw for a valuation tree. For each variable x_i of φ we have a component

- $C_{\exists x_i}$, whose M_φ-implementations have no "witness" for $\exists x_i$, i.e., no may transitions on the top of the diamond encoding the quantifier
- C_{x_i}, whose M_φ-implementations have a path on which there is some v_{x_i} spike but where, on that same path, a $v_{\neg x_i}$-transition occurs subsequently
- $C_{\neg x_i}$, whose M_φ-implementations have a path on which there is some $v_{\neg x_i}$ spike but where, on that same path, a v_{x_i}-transition occurs subsequently.

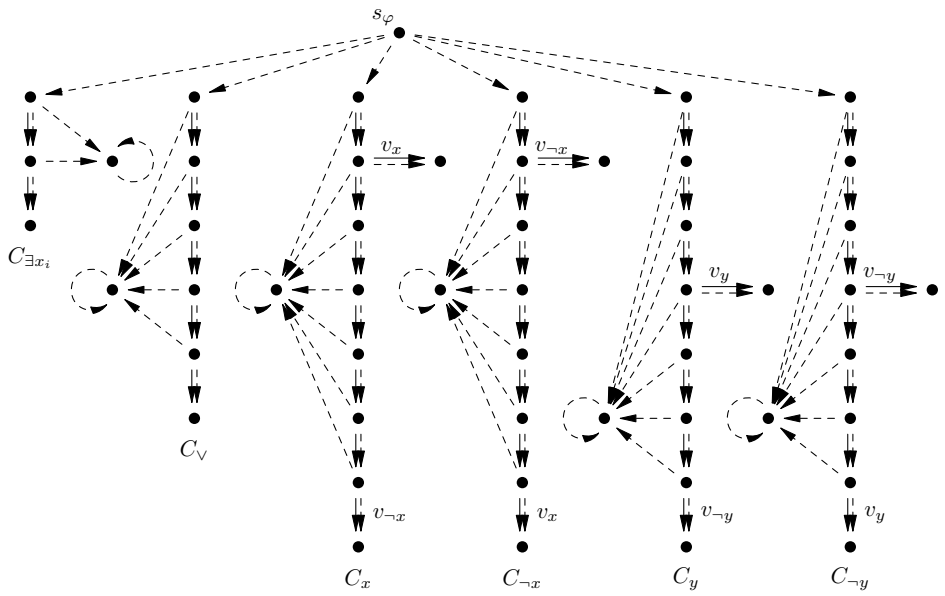


Figure E.13: Modal specification (M_φ, s_φ) for $\varphi = \forall x\, \exists y\, (\neg x \vee y) \wedge (\neg y \vee x)$. All incoming and outgoing transitions of all loop states are labeled with Σ_φ (omitted for clarity)

Finally there is a component $C_\vee$ whose M_φ-implementations all have a path of 3n $\star$-transitions to a dead-end state, and so no such implementation can encode all disjunctions of φ correctly.

Based on the constructions we can present the following theorem.

Theorem 5 *Thorough refinement between modal specifications is PSPACE-hard in the size of these specifications.*

Since the modal transition systems N_φ and M_φ can be constructed in polynomial time in the size of φ, it suffices to show that (E.3) holds.

Note that, by construction, $((\{s_\varphi\}, \emptyset, \emptyset), s_\varphi)$ is an implementation of (M_φ, s_φ) but not of (N_φ, t_φ). So the result also applies to strict thorough refinement.

Corollary 6 *Strict thorough refinement, whether $I(N,t) \subset I(M,s)$, is PSPACE-hard in $|M|$ and $|N|$ for modal and thus also for mixed specifications.*

7 Discussion

First, we relate our results to the complexity of related problems. Second, we discuss and derive our upper bounds.

In [GJ03] efficient translations are given between various classes of 3-valued models such that these translations preserve and reflect the respective refinement notions. These classes of models are all consistent and one of them subsumes

modal transition systems. Therefore our complexity results for common refinement and thorough refinement for modal transition systems transfer to these model classes if we define our three concepts in the same manner for each respective notion of refinement. In particular, our complexity results apply to partial Kripke structures and Kripke modal transition systems.

It is likely that our results extend to "weak" refinement notions that generalize weak bisimulation. This, however, requires a further study. Such refinement notions were systematically studied in [LNW07c] (Paper D).

The "conjunction" gadget used in reducing the common implementation problem for modal transition systems to consistency of a mixed transition system (Section 5) is able to identify states uniquely based on the may/must pattern of transitions encountered en route from the initial state. Nominals, used in hybrid logic [FdR06], are a well known mechanism for identifying states uniquely. One can show NP-hardness of the common implementation problem for *two* modal transition systems already if such systems are enriched with nominals [Ant07].

If specifications are "closed under negation" in that $\neg(M, s)$ has the complement of $I(M, s)$ as set of implementations, then thorough refinement reduces to common implementation: $(M, s) \prec_{th} (N, t)$ is false iff (M, s) and $\neg(N, t)$ have a common implementation. From the results in [Hut05a] it follows easily that modal transition systems do not have such a negation. Support of negation for specifications should require more structure than that found in mixed transition systems. Another open problem is whether non-empty languages $I(M, s)$ accepted by mixed specifications (M, s) can also be accepted by modal specifications; in other words—if a mixed specification is consistent, is it refinement-equivalent to a modal specification?

Generalized model checking [BG00] considers judgments $\mathsf{GMC}(M, s, \varphi)$ which are fossacstrue iff there is an implementation of (M, s) that satisfies φ. For pointed modal specifications (M, s) and Hennessy-Milner formulae φ this is PSPACE-complete in the size of φ [BG00, GJ03]. For each such φ there are $1 \leq m < \infty$ pointed modal specifications (M_i, s_i) such that $\mathsf{GMC}(M, s, \varphi)$ is false iff $I(M, s) \subseteq \bigcup_{i=1}^{m} I(M_i, s_i)$ [Hut05a]. Intuitively, the union on the right-hand side is the set of implementations that satisfy $\neg\varphi$. In general, $m > 1$ so there seems to be no natural and direct reduction of generalized model checking to thorough refinement. For φ in CTL, $\mathsf{GMC}(M, s, \varphi)$ is EXPTIME-complete [BG00, GJ03] but $1 < m$ or $m = \infty$ may hold.

We finally discuss what upper bounds we can provide for the decision problems presented in this paper. Mixed and modal specifications (M, s) have characteristic formulæ $\Psi_{(M,s)}$ [Hut05a] in the modal μ-calculus such that pointed labeled transition systems (L, l) are implementations of (M, s) iff (L, l) satisfies $\Psi_{(M,s)}$. The common implementation and consistency problem reduce to satisfiability checks of $\bigwedge_i \Psi_{(M_i,s_i)}$ and $\Psi_{(M,s)}$, respectively. The thorough refinement problem of whether $(M, s) \prec_{th} (N, t)$ reduces to a validity check of $\neg\Psi_{(N,t)} \vee \Psi_{(M,s)}$.

Validity checking of such vectorized modal μ-calculus formulæ is in EXPTIME

(an unpublished popular wisdom, for which we give a formal argument here). One way in which this membership in EXPTIME can be seen is by translating the problem into alternating tree automata. It is well known that formulæ $\Psi_{(M,s)}$ can be efficiently translated [Wil01] into alternating tree automata $\mathsf{A}_{(M,s)}$ (with parity acceptance condition) that accept exactly those pointed labeled transition systems that satisfy $\Psi_{(M,s)}$. Since non-emptiness, intersection, and complementation of languages is in EXPTIME for alternating tree automata, we get our EXPTIME upper bounds if these automata have size polynomial in $|M|$. Since the size of $\Psi_{(M,s)}$ may be exponential in $|M|$ we require a direct translation from (M, s) into a version of $\mathsf{A}_{(M,s)}$. The formulæ $\Psi_{(M,s)}$ can be written as a system of recursive equations [Lar89] $X_s = \mathit{body}_s$ for each state s of M. We can therefore construct all $\mathsf{A}_{(M,s)}$ in a compositional manner: whenever X_s refers in its body_s to some X_t, then $\mathsf{A}_{(M,s)}$ has a transition to the initial state of $\mathsf{A}_{(M,t)}$ at that point. This $\mathsf{A}_{(M,s)}$ generates the same language as the one constructed from $\Psi_{(M,s)}$, by appeal to the existence of memoryless winning strategies in parity games. The system of equations is polynomial in $|M|$, and so the compositional version of $\mathsf{A}_{(M,s)}$ is polynomial in the size of that system of equations. We summarize:

Theorem 7 *The common implementation, consistency and thorough refinement problems are all in EXPTIME for modal and mixed specifications.*

8 Conclusion

We studied modal and mixed specifications and their fundamental decision problems: consistency (a form of realizability), common implementations (a conjunctive form of consistency), and thorough refinement (a form of implication) of specifications. We established that all these decision problems are in EXPTIME and PSPACE-hard for mixed as well as for modal specifications – keeping in mind that all modal specifications are consistent by construction. These results showed that some of these decision problems are at least as hard as others studied here. This raises the question of whether they in fact have the same complexity.

Table E.1: Tabular summary of the results provided in this paper

	Modal specifications	Mixed specifications
Common implementation	PSPACE-hard, EXPTIME	PSPACE-hard, EXPTIME
Consistency	trivial	PSPACE-hard, EXPTIME
Thorough refinement	PSPACE-hard, EXPTIME	PSPACE-hard, EXPTIME

Acknowledgments.

Harald Fecher made us aware of the counterexample for incompleteness of refinement used in this paper. This then led to the rediscovery of a history of such counterexamples. Nir Piterman helped in improving the presentation of the proof for Theorem 5. We thank Igor Walukiewicz, Wolfgang Thomas and Dietmar Berwanger for independently confirming that validity of vectorized μ-calculus formulæ is in EXPTIME. The referees' comments helped with improving the presentation of this paper.

9 Appendix

9.1 Common Implementation

Let us sketch an argument that the reduction presented in the proof of Theorem 1 is correct. The claim is:

> There exists a winning strategy for Player 1 in GenGeo with graph $G = (V, E, v_0)$ iff there exists a common implementation for

$$\left(\bigcup_{i=0..2} \{(P_i, s_i)\} \right) \cup \bigcup_{e \in E} \left(\{(M_e, s_e), (N_e, t_e)\} \cup \bigcup_{f \in Follow(e) \setminus \{e\}} \{(M_{ef}, s_{ef})\} \right) \tag{E.1}$$

($\Rightarrow$) Consider the implication from former to latter first. Assume we have a winning strategy for Player 1 in the game. We shall construct a common implementation for the specifications mentioned in (E.1). A winning strategy in the generalized geography game can be represented as a game tree, in which each node represents choices by one of the players. Odd nodes on any path from the root to leaves represent moves of Player 1 as suggested by the strategy, while even nodes represent choices of Player 2. Arcs in the tree are labeled with edges from E according to the rules of the game:

1° For any path in the tree from the root to a leaf, any arc label occurs at most once on that path.

2° For any non-root node n if e labels the unique arc incoming into n and f labels any of the outgoing arcs then $\mathbf{src}\, f = \mathbf{tgt}\, e$.

3° Arcs outgoing from the root note are labelled only by edges from *Init*.

Since this is a strategy for Player 1 we have that there is at most one arc outgoing from every Player 1 node. Also, for every Player 2 node n with e labeling the unique arc incoming in n, and for every edge f outgoing from $\mathbf{tgt}\, e$ in G, we have an arc outgoing from n and labeled by f, unless f has already been

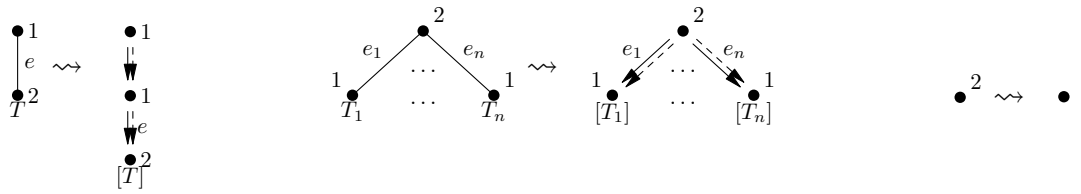


Figure E.14: Rewriting system translating winning strategies for Player 1 into a common implementation

used on the path from root to n. We have all these arcs from Player 2 nodes since the game tree represents a *winning* strategy for Player 1, so all legal game moves of Player 2 have to be present. For the same reason, there is always an arc outgoing from every Player 1 node.

Since the graph G is finite and with every move edges are removed from it, the strategy tree is finite. It is also deterministic. We would like to prove that this tree is actually the common implementation that we are searching for (or basically it is isomorphic to a common implementation). Unfortunately this cannot be claimed directly, because of the introduction of intermediate $\star$-transitions in the reduction. Instead we transform the strategy tree into a modal transition system S as follows.

For each Player 2 node in the tree create a state in the modal transition system. For each Player 1 node n create two states in the modal transition system, n and n', and a single $\star$-transition between the two: $n \xrightarrow{\star} n'$. It is then not hard to specify all the refinement relations that, together, witness the fact that P is a common implementation of all specifications in (E.1).

The following lemma is instrumental in the proof of the other direction:

Lemma 1 *Let $\{(M_l, s_l) \mid 1 \leq l \leq k\}$, with $k > 1$, be a finite family of deterministic (finite-state) modal specifications over the same action alphabet Σ. If all (M_l, s_l) have a common implementation, then they also have a deterministic common implementation.*

Proof 5 (of Lemma 1) *(sketch) Determinize the common implementation I. The states in the determinized system will be subsets of states of I. Then argue that for all i in each such set and for each l there exists such s'_l, a state of M_l, so that $(M_l, s'_l) \prec (Det_I, i)$. This can be done by induction on the size of the I set, following the determinization rules. Then construct the refinement between $Det_{(I,i)}$ and (M_l, s_l) by lifting the original refinement to sets of states.*$\Box$

Lemma 1 can also be shown by noting that the witnessing common implementation computed by the algorithm in [HH04, HH06] is deterministic under the assumptions of this lemma.

($\Leftarrow$) Given a common refinement (I, i) of specifications in (E.1) we will show how this implementation gives rise to a winning strategy for Player 1 in the Generalized Geography game. Without loss of generality, appealing to Lemma 1, we can assume that (I, i) is deterministic.

We begin the game in the initial state of I, with a Player 1 move. There are two simple principles to observe given a current state i:

R1. Player 1's turn: follow one of the outgoing transitions from the current state to a new state i': $i \xrightarrow{\star} \xrightarrow{e} i'$. Play edge e in the game.

R2. Player 2's turn: let Player 2 choose an edge e and then move the implementation from the current state i to a new state i': $i \xrightarrow{e} i'$.

Observe that rule R1 above advances the current implementation from a Player 1 state to a Player 2 state, while rule R2 advances the current implementation from a Player 2 state to a Player 1 state. These two sets of states must necessarily be disjoint as caused by $(P_2, s_2) \prec (I, i)$. In any Player 1 state there is a required $\star$-transition, while such a transition is not allowed in any Player 2 state.

It is easy to prove that, because of the determinism of all systems involved, each sequence of game moves made according to the above rules R1 or R2 moves both the implementation and all specifications of (E.1) deterministically. In other words, it is not possible for I to fulfill requirements of must-transitions of various systems by following a different implementation transition in the same step.

There are several claims that need to be made in order to argue that the above strategy is winning for Player 1. We only discuss the third one in somewhat greater detail

Claim 1. All moves in rule R1 are legal according to the game rules – by appealing to the refinement of all (P_0, s_0), $(M_e t_e)$, $(N_e t_e)$ and $(M_{ef} s_{ef})$.

Claim 2. In rule R1 there is always at least one sequence $\xrightarrow{\star} \xrightarrow{e}$ to follow, i.e. the strategy is winning for Player 1. This is argued by observing that any Player 1 state in I must refine initial states of P_1 and P_2.

Claim 3. All Player 2 moves in rule R2 can actually be followed by the implementation.

The last claim holds because $(M_{e,f} s_{e,f}) \prec (I, i)$ for all $f \in Follow(e) \setminus \{e\}$). Proof by contradiction: Assume that this is not the case. Then there exists a trace of moves $\sigma = \star e_1 f_1 \ldots \star e_n$ so that $i \xrightarrow{\sigma} {}^* i'$ and an edge $f_n \in Follow(e_n)$ such that $f_n \notin \sigma$ and $i' \not\xrightarrow{f_n}$. But then executing σ in s_{ef} deterministically leads to a state s' which must be refined by i' – due to the determinism of I and (M_{ef}). Since f_n is a required transition in that state, we have a contradiction with $(M_{ef}, s_{ef}) \prec (I, i)$. $\square$

9.2 Thorough refinement between modal specifications

Proof 6 (of Theorem 5) *Since the modal transition systems N_φ and M_φ can be constructed in polynomial time in the size of φ of QCNF, it suffices to show that $I(N_\varphi, t_\varphi) \subseteq I(M_\varphi, s_\varphi)$ is equivalent to φ being false.*

1° *Let $I(N_\varphi, t_\varphi) \subseteq I(M_\varphi, s_\varphi)$. Proof by contradiction: assume that φ is true.*

Then there is a valuation tree T such that $T \models \varphi$. We convert T into an implementation of (N_φ, t_φ) that is not an implementation of (M_φ, s_φ), securing the desired contradiction. Valuation tree T is being converted into a modal specification (D, d). Fig. E.15 shows the inductive definition of (D, d), noting that the only source of may transitions stems from the compilation of the propositional CNF kernel. From (D, d) we construct an implementation (E, e) of (N_φ, t_φ). The implementation (E, e) equals (D, d) except that all instances of $[\chi]$ in (D, d) are implemented as follows. Each such instance stems from a leaf V of valuation tree T. Since $T \models \varphi$ we know that for each clause C_j of χ we can pick one literal l_j that is true according to model V. Implement the may v_{l_j}-transition and remove all other may transitions for clause C_j.

The labeled transition system (E, e) is an implementation of (N_φ, t_φ) since the initial quantifier segment of (E, e) and its CNF kernel are matched (up to refinement steps) in the corresponding segments of (E, e).

We argue that (E, e) is not an implementation of (M_φ, s_φ) by considering the possible implementations of the latter modal specification. Let (L, l) be an implementation of (M_φ, s_φ). If (L, l) satisfies

$$\Psi_{stop} = \bigwedge_{\alpha \in \Sigma} [\alpha]\mathsf{ff} \tag{E.4}$$

then it can't be bisimilar to (E, e) since the latter has successor states. So we can subsequently assume that (L, l) does not satisfy Ψ_{stop}. Then there is some transition $(l, \star, l')$ such that (L, l') is bisimilar to one of the components of (M_φ, s_φ). Let e' be the unique successor state of e in E. It remains to show that (E, e') is not bisimilar to (L, l'). We do this by showing that there is some formula of Hennessy-Milner logic that is satisfied by (L, l') but not by (E, e'). Below we write $\langle\star\rangle^k$ for the k-fold nesting of $\langle\star\rangle$. We need a case analysis over which components (L, l') implements.

- *Let (L, l') be an implementation of component $C_{\exists x_i}$. Then (L, l') satisfies*

$$\Psi_{\exists x_i} = \langle\rangle \star^{3i} \Psi_{stop} \tag{E.5}$$

but (E, e') does not since it has a "witness" for that existential choice. In fact, all maximal paths in E from e' have length $3n + 3$.

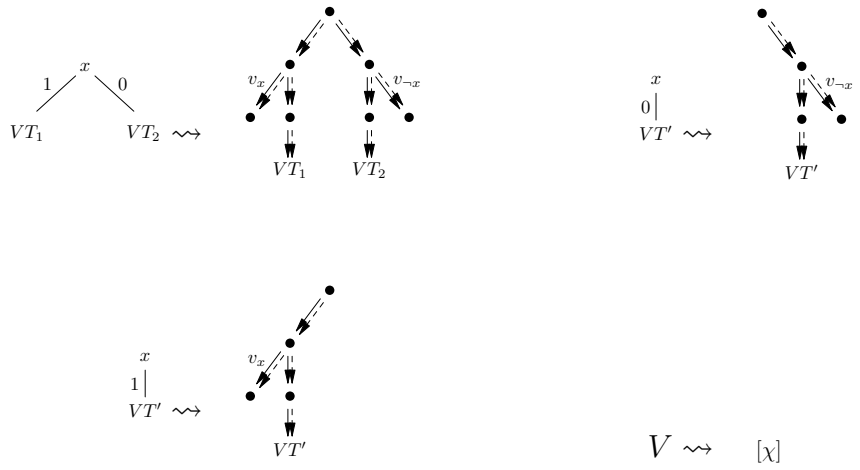


Figure E.15: Deterministic rewrite system that transforms any valuation tree T for $\varphi = Qx_1 \ldots Qx_n.\chi$ into a modal specification (D, d) – with the addition of a topmost must $\star$-transition from d to the topmost state of the rewritten valuation tree. Valuation trees VT', VT_1, and VT_2 all have CNF kernel χ. All leaves V are rewritten to an identical modal specification, namely $[\chi]$ as defined in Fig. E.11. The information encoded in V sets is lost during this rewriting – it will be later on recovered in transforming (D, d) into an implementation (E, e) of (N_φ, t_φ)

- *Let (L, l') be an implementation of component $C_\vee$. Then (L, l') satisfies*

$$\Psi_\vee = \langle\rangle \star^{3n+1} \Psi_{stop} \tag{E.6}$$

  but (E, e') does not since, as already mentioned, all maximal paths in E from e' have length $3n + 3$.

- *Let (L, l') be an implementation of component C_{x_i}. Then (L, l') satisfies*

$$\Psi_{x_i} = \langle\rangle \star^{3i-1} \big(\langle v_{x_i}\rangle \mathsf{tt} \wedge \langle\rangle \star^{3(n-i)+1} \langle v_{\neg x_i}\rangle true\big) \tag{E.7}$$

  but (E, e') does not. This is so since each path in E from e' meets the implementation of the CNF kernel χ and that implementation is consistent with all truth values encountered en route, by construction.

- *Let (L, l') be an implementation of component $C_{\neg x_i}$. Then (L, l') satisfies*

$$\Psi_{\neg x_i} = \langle\rangle \star^{3i-1} \big(\langle v_{\neg x_i}\rangle \mathsf{tt} \wedge \langle\rangle \star^{3(n-i)+1} \langle v_{x_i}\rangle \mathsf{tt}\big) \tag{E.8}$$

  but (E, e') does not, for the same reasons cited in the previous item.

2° *Now let φ be false.* Proof by contradiction*: assume that we have $I(N_\varphi, t_\varphi) \not\subseteq I(M_\varphi, s_\varphi)$. Then there is an implementation (K, k) of (N_φ, t_φ) that is not*

an implementation of (M_φ, s_φ)*. Without loss of generality, we may assume that* (K, k) *is a labeled transition tree. The nature of* (N_φ, t_φ) *then implies that* (K, k) *is a finite labeled transition tree.*

We make further inferences about (K, k)*. The modal specification* (M_φ, t_φ) *is constructed in such a manner that any pointed labeled transition system* (L, l) *that satisfies any of the aforementioned formulæ* Ψ_{stop}*,* $\Psi_{\exists,i}$*,* $\Psi_\vee$*,* Ψ_{x_i} *or* $\Psi_{\neg x_i}$ *implements* (M_φ, t_φ)*. So by assumption,* (K, k) *does not satisfy any of these formulæ. Combining this with the fact that* (K, k) *implements* (N_φ, t_φ) *we infer that*

- *each maximal path from* k *in* K *has length* $3n + 4$
- *the representations of valuations of variables* x_i *are consistent along all paths.*

We use this knowledge to convert this labeled transition tree (K, k) *into a valuation tree* T *such that* $T \models \varphi$*, arriving at the desired contradiction. This transformation should be the "inverse" of that which mapped* T *into* (E, e) *in the previous proof item. In doing this we need to consider two aspects. First,* (K, k) *may have two (or more, as discussed in the next aspect) existential choices for* $\exists x_i$*. Second, as* (K, k) *is characterized only up to bisimulation we may have duplicated paths. Both concerns are addressed by pruning paths, and pruning such paths will maintain that all aforementioned formulæ are still false in the pruned models. This pruning will ensure that we have two choices for* $\forall x_i$ *and one choice for* $\exists x_i$*, resulting in the labeled transition tree* (K', k)*. Each path in this pruned* K' *reaches a "monomial" implementation of the CNF kernel* χ*. We replace this implementation, in situ, with the set of variables that occur in this "monomial". Having done this, we reverse the top three transformations in Fig. E.15 to obtain a potential valuation tree* T *for* φ*. This* T *is not flawed since* (K', k) *is not an implementation of* (M_φ, t_φ)*. This is the desired contradiction to* φ *being false.* $\square$

Bibliography

[AdAdS+06] B. Thomas Adler, Luca de Alfaro, Leandro Dias da Silva, Marco Faella, Axel Legay, Vishwanath Raman, and Pritam Roy. Ticc: A tool for interface compatibility and composition. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 59–62. Springer, 2006.

[AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 71–80, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[AH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120, Vienna, Austria, September 2001. ACM Press.

[AH04] Luca de Alfaro and Thomas A. Henzinger. Interface-based design. In *In Engineering Theories of Software Intensive Systems, Marktoberdorf Summer School*. Kluwer Academic Publishers, 2004.

[AHKV98] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Vardi. Alternating refinement relations. In Davide Sangiorgi and Robert de Simone, editors, *Proceedings of the Ninth International Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *LNCS*, pages 163–178. Springer, 1998.

[AHL+08] Adam Antonik, Michael Huth, Kim G. Larsen, Ulrik Nyman, and Andrzej Wąsowski. Complexity of decision problems for mixed and modal specifications. In *Foundations of Software Science and Computational Structures, FoSSaCS 2008, Proceedings*, volume 4962 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2008. **Paper E in this thesis**.

[AHS02] L. de Alfaro, T. A. Henzinger, and M. I. A. Stoelinga. Timed interfaces. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *EM-*

SOFT 02: 2nd Intl. Workshop on Embedded Software, LNCS, pages 108–122. Springer, 2002.

[Ant07] Adam Antonik. MPhil/PhD transfer report, January 2007. Imperial College London, United Kingdom.

[App98] Andrew A. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[Bat05] Don S. Batory. Feature models, grammars, and propositional formulas. In J. Henk Obbink and Klaus Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[BCH05] Dirk Beyer, Arindam Chakrabarti, and Thomas A. Henzinger. Web service interfaces. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 148–159, New York, NY, USA, 2005. ACM.

[BCU06] Greg Brunet, Marsha Chechik, and Sebastián Uchitel. Properties of behavioural model merging. In Misra et al. [MNS06], pages 98–114.

[BČVZ05] Luboš Brim, Ivana Černá, Pavlína Vařeková, and Barbora Zimmerova. Component-interaction automata as a verification-oriented component-based system specification. In *SAVCBS '05: Proceedings of the 2005 conference on Specification and verification of component-based systems*, page 4, New York, NY, USA, 2005. ACM.

[Ber00] Gérard Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction. Essays in Honour of Robin Milner*, pages 425–454. The MIT Press, Cambridge, MA, 2000.

[BFK+99] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. Pulse: a methodology to develop software product lines. In *SSR '99: Proceedings of the 1999 symposium on Software reusability*, pages 122–131, New York, NY, USA, 1999. ACM.

[BG00] Glenn Bruns and Patrice Godefroid. Generalized model checking: Reasoning about partial state spaces. In Catuscia Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2000.

[BHLP06] Stan Bühne, Günter Halmans, Kim Lauenroth, and Klaus Pohl. Scenario-based application requirements engineering. In Timo

Käkölä and Juan C. Dueñas, editors, *Software Product Lines*, pages 161–194. Springer, 2006.

[BL90] Gérard Boudol and Kim Guldstrand Larsen. Graphical versus logical specifications. In André Arnold, editor, *CAAP*, volume 431 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 1990.

[BL92] Gérard Boudol and Kim Guldstrand Larsen. Graphical versus logical specifications. *Theor. Comput. Sci.*, 106(1):3–20, 1992.

[BLS93] Anders Børjesson, Kim Guldstrand Larsen, and Arne Skou. Generality in design and compositional verification using tav. In *FORTE '92 Proceedings*, pages 449–464, Amsterdam, The Netherlands, The Netherlands, 1993. North-Holland Publishing Co.

[BLS95] Anders Børjesson, Kim Guldstrand Larsen, and Arne Skou. Generality in design and compositional verification using tav. *Formal Methods in System Design*, 6(3):239–258, 1995.

[Bos99] J. Bosch. Product-line architectures in industry: a case study. *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 544–554, 1999.

[Bru97] Glenn Bruns. An industrial application of modal process logic. *Sci. Comput. Program.*, 29(1-2):3–22, 1997.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[CA05] Krzysztof Czarnecki and Michał Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In R. Glück and M. Lowry, editors, *Generative Programming and Component Engineering (GPCE)*, volume 3676 of *LNCS*, pages 422–437. Springer, 2005.

[CD97] C. Weise and D. Lenzkes. Weak refinement for modal hybrid systems. In O. Maler, editor, *Hybrid and Real-Time Systems*, pages 316–330, Grenoble, France, 1997. Springer Verlag, LNCS 1201.

[CdAHS03] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. I. A. Stoelinga. Resource interfaces. In R. Alur and I. Lee, editors, *EMSOFT 03: 3rd Intl. Workshop on Embedded Software*, LNCS. Springer, 2003.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[Cer05] Maura Cerioli, editor. *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3442 of *Lecture Notes in Computer Science*. Springer, 2005.

[CFN03] Cyril Carrez, Alessandro Fantechi, and Elie Najm. Behavioural contracts for a sound composition of components. In Hartmut König, Monika Heiner, and Adam Wolisz, editors, *FORTE 2003 Proceedings*, volume 2767 of *LNCS*, pages 111–126. Springer-Verlag, Berlin, Germany, September 2003.

[CFN05] Cyril Carrez, Alessandro Fantechi, and Elie Najm. Assembling components with behavioural contracts. *Annales des Télécommunications*, 60(7-8):989–1022, 2005.

[ČGL93] Kārlis Čerāns, Jens Chr. Godskesen, and Kim Guldstrand Larsen. Timed modal specification - theory and tools. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 253–267, London, UK, 1993. Springer-Verlag.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[CV07] Luís Caires and Vasco Thudichum Vasconcelos, editors. *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, volume 4703 of *Lecture Notes in Computer Science*. Springer, 2007.

[ČVZ06] Ivana Černá, Pavlína Vařeková, and Barbora Zimmerová. Component substitutability via equivalencies of component-interaction automata. In *FACS'06*, pages 115–130, September 2006. To be published in ENTCS.

[CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 23–34, 10-14 Sept. 2007.

[CWD+06] Zhenbang Chen, Ji Wang, Wei Dong, Zhichang Qi, and W. L. Yeung. An interface theory based approach to verification of web services. In *COMPSAC '06: Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, pages 139–144, Washington, DC, USA, 2006. IEEE Computer Society.

[dAdSF+05] Luca de Alfaro, Leandro Dias da Silva, Marco Faella, Axel Legay, Pritam Roy, and Maria Sorea. Sociable interfaces. In Bernhard Gramlich, editor, *FroCos*, volume 3717 of *Lecture Notes in Computer Science*, pages 81–105. Springer, 2005.

[Dam96] Dennis Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, July 1996.

[DFFU07] Nicolás D'Ippolito, Dario Fishbein, Howard Foster, and Sebastian Uchitel. Mtsa: Eclipse support for modal transition systems construction, analysis and elaboration. In *eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 6–10, New York, NY, USA, 2007. ACM.

[DGG97] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

[DGT96] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation*, volume 1110 of *LNCS*, Dagstuhl Castle, Germany, February 1996. Springer.

[EG98] Sandro Etalle and Maurizo Gabbrieli. Partial evaluation of concurrent constraint languages. *ACM Computing Surveys*, 30(3es), September 1998.

[EMCGP99] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[FdR06] Massimo Franceschet and Maarten de Rijke. Model checking hybrid logics (with an application to semistructured data). *J. Applied Logic*, 4(3):279–304, 2006.

[FG07] Alessandro Fantechi and Stefania Gnesi. A behavioural model for product families. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 521–524, New York, NY, USA, 2007. ACM.

[FH06] Harald Fecher and Michael Huth. Ranked predicate abstraction for branching time: Complete incremental, and precise. In *ATVA*, volume 4218 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2006.

[FUB06] Dario Fischbein, Sebastian Uchitel, and Victor Braberman. A foundation for behavioural conformance in software product line architectures. In *ROSATEA '06 Proceedings*, pages 39–48, New York, NY, USA, 2006. ACM Press.

[GHJ01] Patrice Godefroid, Michael Huth, and Radha Jagadeesan. Abstraction-based model checking using modal transition systems. *Lecture Notes in Computer Science*, 2154:426+, 2001.

[GHW85] J. V. Guttag, J. J. Horning, and J. M. Wing. The larch family of specification languages. *IEEE Softw.*, 2(5):24–36, 1985.

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GJ02] Patrice Godefroid and Radha Jagadeesan. Automatic abstraction using generalized model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 137–150. Springer, 2002.

[GJ03] Patrice Godefroid and Radha Jagadeesan. On the expressiveness of 3-valued models. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *VMCAI*, volume 2575 of *Lecture Notes in Computer Science*, pages 206–222. Springer, 2003.

[Gom01] Hassan Gomaa. *Design Software Product Lines with UML*. Addison-Wesley, 2001.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HH04] Altaf Hussain and Michael Huth. On model checking multiple hybrid views. Technical report, Department of Computer Science, University of Cyprus, 2004. TR-2004-6.

[HH06] Altaf Hussain and Michael Huth. Automata games for multiple-model checking. *Electr. Notes Theor. Comput. Sci.*, 155:401–421, 2006.

[HJS01] Michael Huth, Radha Jagadeesan, and David Schmidt. Modal transition systems: A foundation for three-valued program analysis. *Lecture Notes in Computer Science*, 2028, 2001.

[HKY96] Haruo Hosoya, Naoki Kobayashi, and Akinori Yonezawa. Partial evaluation scheme for concurrent languages and its correctness. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96*, volume 1123 of *LNCS*, pages 625–632. Springer, 1996.

[HL89] Hans Hüttel and Kim Guldstrand Larsen. The use of static constructs in a modal process logic. In *LFCS: The 1st International Symposium on Logical Foundations of Computer Science*, 1989.

[HM85] Matthew Henessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of ACM*, pages 137–161, 1985.

[HM06] Thomas A. Henzinger and Slobodan Matic. An interface algebra for real-time components. In *IEEE Real Time Technology and Applications Symposium*, pages 253–266. IEEE Computer Society, 2006.

[HMT99] John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation: Practice and Theory. International Summer School*, volume 1706 of *LNCS*. Springer, 1999.

[HMU01] John E. Hopcroft, Rejeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 2nd edition, 2001.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.

[HRS05] Holger Hermanns, Jakob Rehof, and Marielle I. A. Stoelinga, editors. *Workshop Procedings FIT 2005: Foundations of Interface Technologies*, ENTCS. Elsevier Science Publishers, 2005.

[Hüt88] Hans Hüttel. Operational and denotational properties of modal process logic. Master's thesis, Computer Science Department. Aalborg University, 1988.

[Hut02] Michael Huth. Model checking modal transition systems using kripke structures. In Agostino Cortesi, editor, *VMCAI*, volume 2294 of *Lecture Notes in Computer Science*, pages 302–316. Springer, 2002.

[Hut05a] Michael Huth. Labelled transition systems as a Stone space. *Logical Methods in Computer Science*, 1(1):1–28, January 2005.

[Hut05b] Michael Huth. Refinement is complete for implementations. *Formal Asp. Comput.*, 17(2):113–137, 2005.

[IAR] IAR visualSTATE®. `www.iar.com/Products/VS`.

[IK01] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *POPL 2001*. ACM Press, 2001.

[ILO] Ilog CPLEX. `www.ilog.com/products/cplex`.

[JB02] Michel Jaring and Jan Bosch. Representing variability in software product lines: A case study. In *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, pages 15–36, London, UK, 2002. Springer-Verlag.

[JB04] Michel Jaring and Jan Bosch. Architecting product diversification - formalizing variability dependencies in software product family engineering. In *QSIC '04: Proceedings of the Quality Software, Fourth International Conference*, pages 154–161, Washington, DC, USA, 2004. IEEE Computer Society.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[JL91] Bengt Jonsson and Kim Guldstrand Larsen. On the complexity of equation solving in process algebra. In Samson Abramsky and T. S. E. Maibaum, editors, *TAPSOFT, Vol.1*, volume 493 of *Lecture Notes in Computer Science*, pages 381–396. Springer, 1991.

[JP01] Bart Jacobs and Erik Poll. A logic for the java modeling language jml. In *FASE '01: Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, pages 284–299, London, UK, 2001. Springer-Verlag.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[KE03] Chethana Kuloor and Armin Eberlein. Aspect-oriented requirements engineering for software product lines. In *ECBS*, pages 98–107. IEEE Computer Society, 2003.

[Lar86] Kim G. Larsen. *Context Dependent Bisimulation Between Processes*. PhD thesis, Edinburgh University, 1986.

[Lar87] Kim G. Larsen. A context dependent equivalence between processes. *Theoretical Computer Science*, 49:184–215, 1987.

[Lar89] Kim Guldstrand Larsen. Modal specifications. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 1989.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[Lea96] Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 121–142. Kluwer Academic Publishers, Boston, 1996.

[Lev95] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[LLW05] Kim Guldstrand Larsen, Ulrik Larsen, and Andrzej Wąsowski. Color-blind specifications for transformations of reactive synchronous programs. In Cerioli [Cer05], pages 160–174.

[LM92] Kim G. Larsen and Robin Milner. A compositional protocol verification using relativized bisimulation. *Inf. Comput.*, 99(1):80–108, 1992.

[LMN05] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using UPPAAL. In *Formal Approaches to Testing of Software (FATES), Linz, Austria. September 21, 2004*, volume 1644 of *LNCS*. Springer, 2005.

[LNAH+01] J. Lind-Nielsen, H. R. Andersen, H. Hulgaard, G. Behrmann, K. Kristoffersen, and K. G. Larsen. Verification of large state/event systems using compositionality and dependency analysis. *Formal Methods in System Design*, 18(1):5–23, 2001.

[LNW05] Kim G. Larsen, Ulrik Nyman, and Andrzej Wąsowski. Interface input/output automata: Splitting assumptions from guarantees. In Hermanns et al. [HRS05].

[LNW06a] Kim G. Larsen, Ulrik Nyman, and Andrzej Wąsowski. An interface theory for input/output automata. Technical Report RS-06-11, BRICS, June 2006.

[LNW06b] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wąsowski. Interface input/output automata. In Misra et al. [MNS06], pages 82–97. **Paper B in this thesis**.

[LNW07a] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wąsowski. Modal i/o automata for interface and product line theories. In Nicola [Nic07], pages 64–79. **Paper C in this thesis**.

[LNW07b] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wąsowski. Modeling software product lines using color-blind transition systems. *STTT*, 9(5-6):471–487, 2007. **Paper A in this thesis**.

[LNW07c] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wąsowski. On modal refinement and consistency. In Caires and Vasconcelos [CV07], pages 105–119. **Paper D in this thesis**.

[LSW95] Kim Guldstrand Larsen, Bernhard Steffen, and Carsten Weise. A constraint oriented proof methodology based on modal transition systems. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *TACAS*, volume 1019 of *Lecture Notes in Computer Science*, pages 17–40. Springer, 1995.

[LSW96] K. G. Larsen, B. Steffen, and C. Weise. Fischer's protocol revisited: a simple proof using modal constraints. *Lecture Notes in Computer Science*, 1066:604–615, 1996.

[LT88] Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *LICS*, pages 203–210. IEEE Computer Society, IEEE Computer Society, 1988.

[LW94] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[LX90] Kim Guldstrand Larsen and Liu Xinxin. Equation solving using modal transition systems. In *Fifth Annual IEEE Symposium on Logics in Computer Science (LICS), 4–7 June 1990, Philadelphia, PA, USA*, pages 108–117, 1990.

[LX04] E. A. Lee and Y. Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing Journal*, 2004.

Special issue on Semantic Foundations of Engineering Design Languages.

[Lyn88] Nancy Lynch. I/O automata: A model for discrete event systems. In *Annual Conference on Information Sciences and Systems*, pages 29–38, Princeton University, Princeton, N.J., 1988.

[LZZ05] Edward A. Lee, Haiyang Zheng, and Ye Zhou. Causality interfaces and compositional causality analysis. In Hermanns et al. [HRS05].

[Mai03] Patrick Maier. Compositional circular assume-guarantee rules cannot be sound and complete. In A.D. Gordon, editor, *Foundations of Software Science and Computational Structures: 6th International Conference, FOSSACS 2003*, volume 2620 of *LNCS*, pages 343–357. Springer, 2003.

[Mey92] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25, 1983.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MNS06] Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors. *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*. Springer, 2006.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[Mur95] Masaki Murakami. Partial evaluation of reactive communciating processes using temporal logic formulas. In *Workshop on Algebraic and Object-Oriented Approaches to Software Science*, 1995.

[Nic07] Rocco De Nicola, editor. *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*. Springer, 2007.

[Obj99] Object Management Group. OMG Unified Modelling Language specification, 1999. `www.omg.org`.

[Par76] David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, Vol. SE-2(No. 1):1–9, March 1976.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *Proceedings of 5th GI Conference*, volume 104 of *LNCS*, 1981.

[PBvdL05] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering—Foundations, Principles, and Techniques.* Springer, jul 2005.

[PV02] F. Plasil and S. Visnovsky. Behavior protocols for software components. *Software Engineering, IEEE Transactions on*, 28(11):1056–1076, Nov 2002.

[RR02] Sriram K. Rajamani and Jakob Rehof. Conformance checking for models of asynchronous message passing software. In Ed Brinksma and Kim Guldstrand Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 166–179, Copenhagen, Denmark, July 2002. Springer.

[Sch01] David A. Schmidt. From trace sets to modal transition systems by stepwise abstract interpretation. In Masami Hagiya, Yoshiki Kinoshita, and John Power, editors, *Proceedings Workshop on Structure-Preserving Relations, Amagasaki, Japan.* entcs, 2001. To appear.

[Sch06a] Heiko Schmidt. Comparing disjunctive modal transition systems with an one-selecting variant. In *NWPT'06 – The 18th Nordic Workshop on Programming Theory (NWPT'06) Reykjavík, Iceland, 18-20 October, 2006*, Reykjavík, Iceland, October 2006. Reykjavík University.

[Sch06b] Heiko Schmidt. Comparing disjunctive modal transition systems with their one-selecting variant. Master's thesis, Christian-Albrechts-UniversitÃd't zu Kiel, 2006.

[sco] Scope. `www.itu.dk/~wasowski/scope`.

[SDH00] Kanna Shimizu, David L. Dill, and Alan J. Hu. Monitor-based formal specification of PCI. In *Formal Methods in Computer-Aided Design*, pages 335–353, 2000.

[SF07] Heiko Schmidt and Harald Fecher. Comparing disjunctive modal transition systems with a one-selecting variant. *To appear in the Journal of Logic and Algebraic Programming*, 2007.

[Sun] Sun Microsystems. Java card(TM) specification. `java.sun.com/products/javacard/specs.html`.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[TG06] Jean-Pierre Talpin and Paul Guernic. An algebraic theory for behavioral modeling and protocol synthesis in system design. *Form. Methods Syst. Des.*, 28(2):131–151, 2006.

[Tre96] Jan Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[TWS06] Lothar Thiele, Ernesto Wandeler, and Nikolay Stoimenov. Real-time interfaces for composing real-time systems. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 34–43, New York, NY, USA, 2006. ACM.

[UBC07] Sebastián Uchitel, Greg Brunet, and Marsha Chechik. Behaviour model synthesis from properties and scenarios. In *ICSE*, pages 34–43. IEEE Computer Society, 2007.

[UC04] Sebastián Uchitel and Marsha Chechik. Merging partial behavioural models. In Richard N. Taylor and Matthew B. Dwyer, editors, *SIGSOFT FSE*, pages 43–52. ACM, 2004.

[vG90] Rob van Glabbeek. The linear time–branching time spectrum (extended abstract). In J.C.M. Beaten and J.W. Klop, editors, *Theories of Concurrency: Unification and Extension (CONCUR)*, volume 458 of *LNCS*, pages 278–297. Springer, 1990.

[Wąs03] Andrzej Wąsowski. On Efficient Program Synthesis from Statecharts. In *ACM SIGPLAN Languages, Compilers, and Tools for Embedded Systems (LCTES)*, San Diego, USA, June 2003. ACM Press.

[Wąs04] Andrzej Wąsowski. Automatic generation of program families by model restrictions. In *Software Product Line Conference (SPLC)*, volume 3154 of *LNCS*. Springer, 2004.

[Wąs05] Andrzej Wąsowski. *Code Generation and Model Driven Development for Constrained Embedded Software*. PhD thesis, IT University of Copenhagen, January 2005.

[Wil01] Thomas Wilke. Alternating tree automata, parity games, and modal μ-calculus. *Bull. Soc. Math. Belg.*, 8(2), May 2001.

[Win87] Jeannette M. Wing. Writing larch interface language specifications. *ACM Trans. Program. Lang. Syst.*, 9(1):1–24, 1987.

[WT05] Ernesto Wandeler and Lothar Thiele. Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 80–89, New York, NY, USA, 2005. ACM.

[WT06] Ernesto Wandeler and Lothar Thiele. Interface-based design of real-time systems with hierarchical scheduling. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 243–252, Washington, DC, USA, 2006. IEEE Computer Society.

[Xin92] Liu Xinxin. *Specification and Decomposition in Concurrency*. PhD thesis, Department of Mathematics and Comnputer Science, Aalborg University, April 1992.

[YS97] D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

[ZW97] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Trans. Softw. Eng. Methodol.*, 6(4):333–369, 1997.