University of Bamberg
Faculty of Information Systems and Applied Computer Science
Software Technologies Research Group

Modal Interface Theories for Specifying Component-based Systems

Dissertation Thesis

Author:
Sascha Fendrich

Bamberg, July 20, 2017

Supervisor and first reviewer:	Prof. Dr. Gerald Lüttgen Otto-Friedrich-University Bamberg Germany
Second Reviewer:	Prof. Dr. Rolf Hennicker Ludwig-Maximilians-University Munich Germany
Date of Defense:	July 20, 2017

to Eric
who developed concurrently to this thesis
and safely avoided a critical race condition

Acknowledgements

First of all, I thank my advisor, Gerald Lüttgen, for his invaluable support: Thank you for many hours of fruitful discussions, for your excellent and experienced advice in many situations and for the financial support. I am very grateful that you gave me the opportunity to participate and make experience in different areas of research-relevant duties such as reviewing, paper-writing, teaching, participating in conferences and summer schools. Thank you for the pleasant working atmosphere in the Software Technologies Research Group and for your attitude of making things possible. I am very happy that this project was possible and I much enjoyed working with you as my advisor, boss, colleague and coauthor.

Further, I thank my project associates from Augsburg, Ferenc Bujtor and Walter Vogler, for being my colleagues, coauthors, discussion partners and critical readers; Johannes Gareis for writing his master's thesis in my project and later being my colleague and discussion partner; Michael Mendler and Diedrich Wolter for being members of my 'Promotionskommission' and for their comments and advices on many parts of my work; my second reviewer Rolf Hennicker for his willingness to evaluate my thesis and for discussing it extensively; the attendees of the Monday Afternoon Club for giving me the opportunity to discuss my work and for their helpful suggestions and comments; the organisers, teachers and participants of the Marktoberdorf Summer School 2015 and of the BETTY Summer School 2016 for the great lectures and the fruitful discussions; my office mate Alexander Heußner, for being my colleague, for answering lots of questions and giving me advice, and for the scientific and non-scientific discussions; my colleagues at the Software Technologies Research Group, Alexandra Homer, Thomas Rupprecht, Ons Seddiki, David White, Thomas Wunder and Eugene Yip for the great time, the countless (but countable) lunch break walks, the scientific and non-scientific discussions; my beloved Britta for moving with me to Bamberg, for all the time and effort she spent for commuting, for the Thursday evening meals, for proof reading and discussing my work, for all the patience and, of course, for the wonderful life together; my parents who always encouraged and supported me in my studies and, thus, gave me a great education, without which I would not have been able to accomplish this thesis; Thomas Streicher for teaching me logic and mathematical foundations of computer science when I was a student at Technische Universität Darmstadt and for his comments on negation; Luca Aceto and Kim G. Larsen for their comments on negation; Rance Cleaveland for his comments on quotienting, negation and refinement checking; Jan Křetínský for his support on the quotient; Malte Lochau, Lars Luthmann and Stephan Mennicke for discussions on error states; Uwe Nestmann and Kirstin Peters for discussions on session types and friends; Salomon Sickert for his support on QBF-based refinement checking; André Braun for the friendship and the hospitality he gave to a homeless scientist; the developers of Gnu, Linux, Debian, Haskell, Go, Graphviz, LaTeX, LibreOffice, vi and all other free software I used to write this dissertation; the Deutsche Forschungsgesellschaft for the support provided under the grants LU-1748/3-1&2; the dual career service of the University of Bamberg for the provided support.

Abstract

Large software systems frequently manifest as complex, concurrent, reactive systems and their correctness is often crucial for the safety of the application. Hence, modern techniques of software engineering employ *incremental, component-based approaches* to systems design. These are supported by *interface theories* which may serve as specification languages and as semantic foundations for software product lines, web-services, the internet of things, software contracts and conformance testing. Interface theories enable a systems designer to express *communication requirements* of components on their environments and to reason about the mutual *compatibility* of these requirements in order to guarantee the *communication safety* of the system. Further, interface theories enrich traditional operational specification theories by declarative aspects such as conjunction and disjunction, which allow one to specify systems heterogeneously.

However, substantial practical aspects of software verification are not supported by current interface theories, e.g., reusing components, adapting components to changed operational environments, reasoning about the compatibility of more than two components, modelling software product lines or tracking erroneous behaviour in safety-critical systems. The goal of this thesis is to investigate the theoretical foundations for making interface theories more practical by solving the above issues. Although partial solutions to some of these issues have been presented in the literature, none of them succeeds without sacrificing other desired features. The particular challenge of this thesis is to solve these problems simultaneously within a single interface theory. To this end, the arguably most general interface theory *Modal Interface Automata* (MIA) is extended, yielding the interface theory *Error-preserving Modal Interface Automata* (EMIA).

The above problems are addressed as follows. Quotient operators are adjoint to composition and, therefore, support component reuse. Such a quotient operator is introduced to both MIA and EMIA. It is the first one that considers nondeterministic dividends and compatibility. Alphabet extension operators for MIA and EMIA allow for the change of operational environment by permitting one to adapt system components to new interactions without breaking previously satisfied requirements. Erroneous behaviour is identified as a common source of problems with respect to the compatibility of more than two components, the modelling of software product lines and erroneous behaviour in safety-critical systems. EMIA improves on previous interface theories by providing a more precise semantics with respect to erroneous behaviour based on error-preservation. The relation between error-preservation and the usual error-abstraction employed in previous interface theories is investigated, establishing a Galois insertion from MIA into EMIA that is relevant at the levels of specifications, composition operations and proofs. The practical utility of interface theories is demonstrated by providing a software implementation of MIA and EMIA that is applied to two case studies. Further, an outlook is given on the relation between type checking and refinement checking. As a proof of concept, the simple interface theory *Interface Automata* is extended to a behavioural type theory where type checking is a syntactic approximation of refinement checking.

Zusammenfassung

Große Softwaresysteme bilden häufig komplexe, nebenläufige, reaktive Systeme, deren Korrektheit für die Sicherheit der Anwendung entscheidend ist. Daher setzen moderne Verfahren der Softwaretechnik *inkrementelle, komponentenbasierte Ansätze* zum Software-Entwurf ein. Diese werden von *Interface-Theorien* unterstützt, die als Spezifikationssprachen und semantische Grundlagen für Softwareproduktlinien, Web-Services, das Internet der Dinge, Softwarekontrakte und Konformanztests dienen können. Interface-Theorien ermöglichen es, *Kommunikationsanforderungen* von Komponenten an ihre Umgebung auszudrücken, um die gegenseitige *Kompatibilität* dieser Anforderungen zu überprüfen und die *Kommunikationssicherheit* des Systems zu garantieren. Zudem erweitern Interface-Theorien traditionelle operationale Spezifikationstheorien um deklarative Aspekte wie beispielsweise Konjunktion und Disjunktion, die heterogenes Spezifizieren ermöglichen.

Allerdings werden wesentliche praktische Aspekte der Softwareverifikation von Interface-Theorien nicht unterstützt, z.B. das Wiederverwenden von Komponenten, das Anpassen von Komponenten an geänderte operationale Umgebungen, die Kompatibilitätsprüfung von mehr als zwei Komponenten, das Modellieren von Softwareproduktlinien oder das Zurückverfolgen von Fehlverhalten sicherheitskritischer Systeme. Diese Arbeit untersucht die theoretischen Grundlagen von Interface-Theorien mit dem Ziel, die oben genannten praktischen Probleme zu lösen. Obwohl es in der Literatur Teillösungen zu manchen dieser Probleme gibt, erreicht keine davon ihr Ziel, ohne andere wünschenswerte Eigenschaften aufzugeben. Die besondere Herausforderung dieser Arbeit besteht darin, diese Probleme innerhalb einer einzigen Interface-Theorie zugleich zu lösen. Zu diesem Zweck wurde die wohl allgemeinste Interface-Theorie *Modal Interface-Automata* (MIA) zu der Interface-Theorie *Error-preserving Modal Interface Automata* (EMIA) weiterentwickelt.

Die obigen Probleme werden wie folgt gelöst. Ein zur Komposition adjungierter Quotientenoperator, der das Wiederverwenden von Komponenten ermöglicht, wurde für MIA und EMIA eingeführt. Es handelt sich dabei um den ersten Quotientenoperator, der nichtdeterministische Dividenden und Kompatibilität betrachtet. Alphabeterweiterungsoperatoren erlauben eine Änderung der operationalen Umgebung, indem sie es ermöglichen, Komponenten an neue Interaktionen anzupassen, ohne zuvor erfüllte Anforderungen zu missachten. Fehlerhaftes Verhalten wird als eine gemeinsame Ursache von Problemen bezüglich der Kompatibilität von mehr als zwei Komponenten, der Modellierung von Softwareproduktlinien und des Fehlverhaltens sicherheitskritischer Systeme erkannt. EMIA verbessert bisherige Interface-Theorien durch eine präzisere Fehlersemantik, die auf dem Erhalten von Fehlern beruht. Als Beziehung zwischen diesem Fehlererhalt und der in bisherigen Interface-Theorien üblichen Fehlerabstraktion ergibt sich eine Galois-Einbettung von MIA in EMIA, die auf den Ebenen der Spezifikationen, Operatoren und Beweise relevant ist. Die praktische Anwendbarkeit von Interface-Theorien wird mittels einer Implementierung von MIA und EMIA als Software und deren Anwendung auf zwei Fallstudien demonstriert. Zudem wird das Verhältnis zwischen Verfeinerung und Typprüfung diskutiert. In einer Machbarkeitsstudie wurde die einfache Interface-Theorie *Interface Automata* zu einer Verhaltenstyptheorie erweitert, bei der die Typprüfung eine syntaktische Approximation der Verfeinerung ist.

Contents

List of Figures

List of Tables

1. Introduction

This chapter motivates the subject of this thesis, namely, the investigation of interface theories for designing complex reactive systems and presents the context of the research project within which this thesis has been developed. The thesis is summarised, and a list of contributions is given.

1.1. Motivation and Context

Large software systems have to manage resources, coordinate a multitude of hardware devices, and organise the interaction between software components. Interconnecting such systems of interacting components (whether application-level or OS-level) to even larger *systems of systems* is nowadays standard in many areas, such as cloud computing, automotive, avionics, or industrial automation. As a consequence, today's software frequently manifests as complex, concurrent, reactive systems, and the correctness of the software is often crucial for the safety of the entire application. In order to tackle the complexity of developing such large systems, a common practice of software engineering is to decompose a system into components and to distribute the development work component-wise among different teams or vendors. Such an *incremental, component-based design approach* requires that the components of a system may be implemented independently and the system may be composed from these components successively [AH05; Rac+11].

However, precise reasoning about the complex interactions and the mutual requirements of components to ensure the interoperability of the independent implementations is difficult and requires support of formal methods. Such support is provided by modern, refinement-based approaches to software design. In particular, *interface theories* [AH01a; AH05; Bau+10; Buj+16; BV14; Che+12; FL16b; LNW07; LVF15; Rac+11] have been developed as specification theories that permit one to specify requirements of components on their environment and to check the mutual compatibility of these requirements at the level of the component interfaces. Following [AH05; Rac+11], we discuss properties of interface theories that are desired in practice, and minimum requirements that are necessary to support these properties:

Independent implementability. A *compositional refinement preorder* permits a systems designer to start with coarse specifications of components and to refine them stepwise by taking decisions on *underspecified* properties until an implementation is reached. Compositionality guarantees the substitutivity of these stepwise refinements, i.e., a specification of a component may be substituted by any of its refinements or implementations. Hence, components may be implemented independently, and the *communication safety*

of the component specifications guarantees that the component implementations are also communication safe.

Communication safety. In order to interoperate correctly, components of a system have to mutually meet their communication requirements. These mutual requirements must be preserved by the refinement preorder in order to preserve the communication safety of the system when implementing independently. Communication safety is often reduced to a binary compatibility concept, e.g., [AH05]. However, the communication safety of multi-component assemblies, i.e., compositions with more than two components, is independent of the pairwise compatibility of their components even if the composition operation is associative [HK15]. Hence, a more general approach to communication safety is required.

Incremental design. An interface theory supports incremental design if one may compose a system from components incrementally by starting with a subset of the components to which other components may be added successively. A minimum requirement for supporting incremental design is a binary composition operation, i.e., a parallel operator, that is associative and commutative. In addition, the intermediate results of an incremental composition are *open systems* in the sense that they may have unbound communication channels [Mil80]. Such unbound channels may change the relevance of potential communication errors depending on whether these channels are used in the larger scope of the system, e.g., a communication error may become unreachable when a channel is not used. Hence, interface theories should support an open systems view.

Component reuse. A *quotient operator* that is adjoint to composition complements incremental design by permitting one to reuse components. Starting from a global system specification and some existing components, the quotient operator synthesises a coarsest specification of the remaining parts of a partially implemented system [And95].

Perspective-based specification. Specifying a system component from separate perspectives such that the component satisfies each of these perspective specifications is a common practice in software engineering. For example, each requirement for a component might describe a perspective. The component's overall specification is obtained as the *conjunction* of the perspective specifications. It should be the coarsest specification refining all perspective specifications, i.e., the greatest lower bound with respect to the refinement preorder [LV11]. Such an approach is particularly effective, if a single perspective is only required to cope with the interactions relevant for this perspective. In order to merge the action alphabets of different perspectives, interface theories should provide *alphabet extension operators*.

Nondeterminism and abstraction from internal behaviour. Different behaviours of a system may become indistinguishable when abstracting from implementation/specification details. Therefore, even deterministic systems may appear nondeterministic at a certain

level of abstraction, in particular, in the context of concurrency. When developing a concurrent system, the internal communication between components of a subsystem may be unobservable from outside the subsystem. Hence, a specification theory should provide operators for hiding such internal interactions, and the refinement preorder should allow one to abstract from such hidden actions [Mil80].

Variability of implementations. When distributing the development process among different teams or vendors, one wishes to provide a variability of implementations in order to increase the reusability of a component in different contexts. Implementation variants may be organised into *software product lines* that may be expressed by more abstract specifications, e.g., by employing *underspecification* or *disjunction*. When composing concrete products of different software product lines, one is usually not interested in the overall compatibility of the whole product lines but in finding compatible subfamilies of concrete software products [LNW07].

Behavioural types. When transitioning from design to implementation, it is necessary to consider data. Communication safety then also must include type safety because components may exchange typed data during interaction, i.e., one wishes to prevent that one component tries reading a string when a different one is sending an integer. To this end, an interface theory must support to refine behavioural specifications into behavioural types. In particular, type checking may be a computationally more efficient approximation of refinement checking.

Current languages for specifying complex, concurrent systems [AH01a; AH05; BHW11; Bau+10; Buj+16; BV14; Che+12; CE81; HIJ15; LNW07; LV13a; LVF15; QS82; Rac08; Rac+11] support some of the above properties but are in conflict with other ones. They have several gaps in the mathematical robustness of their semantics, in particular, with respect to concurrency and communication safety, impeding the development of tools that make these theories practical. Examples of such gaps and of theories that are affected by these gaps are (see also Section 2.4):

1. A non-compositional refinement preorder makes interface theories unsuitable for independent implementability, e.g., [AH01a; LNW07];

2. A non-associative composition operator undermines the suitability of interface theories for incremental design, e.g., [Rac+11];

3. Lack of support for nondeterminism and internal behaviour limits the practical applicability of interface theories, e.g., [Rac08; Rac+11] with no support and [AH01a; AH05; LNW07; LV13a] with limited support;

4. Lack of support for checking compatibility of multiple components restricts the guarantees an interface theory can provide with respect to communication safety, e.g., [AH01a; AH05; Bau+10; Buj+16; BV14; Che+12; LNW07; LV13a; LVF15; Rac08; Rac+11];

5. Lack of support of an open systems view restricts the suitability of an interface theory for incremental design, e.g., [Bau+10] and the pessimistic variant of [LVF15];

6. Limited support for reasoning about variability of implementations makes several theories unsuitable for modelling software product lines, e.g., [AH01a; AH05; Bau+10; Buj+16; BV14; Che+12; LNW07; LV13a; LVF15; Rac08; Rac+11];

7. The possibility to introduce unwanted behaviour when refining a system undermines communication safety and is particularly undesired when considering safety-critical systems, e.g., [AH01a; AH05; Buj+16; BV14; Che+12; LNW07; LV13a; LVF15; Rac+11];

8. Lack of support for reasoning about data restricts *all current interface theories* to design and makes them unsuitable for implementation; first steps towards relaxing this limitation are presented in [BHW11; HIJ15];

9. Logic and operational semantics exist on separate levels and cannot be freely intermixed in a truly *heterogeneous specification*, e.g., when checking an operational model against temporal logic formulas in modern verification approaches [CE81; QS82].

The collaborative research project *Foundations of Heterogeneous Specifications Using State Machines and Temporal Logic* supported by the German Research Foundation (DFG; grants No. LU1748/3-1&2 and VO615/12-1&2) investigates the theoretical foundations and the practical utility of languages combining operational and declarative formalisms for specifying concurrent, reactive systems. The aim of the project is to address the abovementioned gaps in such heterogeneous languages. An additional goal of this project, which is beyond the scope of this thesis, is to extend interface theories to support reasoning about further properties such as liveness and fairness.

1.2. Summary and Structure of the Thesis

This thesis presents the interface theory *Error-preserving Modal Interface Automata* (EMIA) that improves on previous interface theories by closing several of the gaps mentioned in Section 1.1. In particular, it provides a more detailed semantics with respect to unwanted behaviour and investigates the relation between this refined semantics and the error-abstracting semantics of previous interface theories.

Chapter 2 of this thesis defines basic concepts and a formal theory of specification (Section 2.1). A summary of existing specification formalisms is presented, and the term 'specification' as used in this thesis is classified into this summary (Section 2.2.1). Existing formalisms for concurrent systems are reviewed (Sections 2.2.2–2.2.7). Special attention is paid to interface theories based on *Interface Automata* (IA) [AH01a] (Section 2.3), and a comparative study of existing specification theories with respect to the features they support is conducted (Section 2.4).

Chapter 3 presents two variants of the interface theory *Modal Interface Automata*, namely Error-preserving Modal Interface Automata (EMIA) and Error-abstracting Modal Interface Automata (MIA). The theory EMIA (Section 3.1) is the main contribution of this thesis. It supports a compositional refinement preorder, the operators parallel composition, hiding, restriction, quotient, conjunction, disjunction and several alphabet extension operators. Due to error-preservation, EMIA has a more detailed semantics with respect to unwanted behaviour, which fixes several shortcomings of previous interface theories. In addition, the logical operators implication and negation are investigated, and it is shown that specification theories based on Modal Transition Systems (MTS) [LT88; LX90] are not closed under these operators. The theory MIA (Section 3.2) has been co-developed with Bujtor, Lüttgen and Vogler [Buj+16]. Historically, it has been devised first. However, in this thesis MIA is presented as an abstraction of EMIA by establishing a Galois insertion between the two theories. In particular, this sheds light on the role of error-abstraction with respect to the weaknesses of error-abstracting theories (Section 3.3). Alternative approaches, possible extensions and the computational complexity of the most important EMIA operators are discussed (Section 3.4).

Chapter 4 presents the incremental, component-based design methodology supported by interface theories. This methodology is then applied to the design of a client-server application within MIA (Section 4.2) and to the design of a railway control system within EMIA (Section 4.3). An overview of software tools implementing specification theories is given, including two tools that have been developed in the context of this thesis (Section 4.4).

Chapter 5 gives an outlook of how interface theories based on IA may be extended to behavioural type theories that support sending and receiving typed messages. A behavioural type system similar to session types [Hon93] is presented, and the standard theorems of subject reduction, congruence and type safety are proved.

Chapter 6 summarises the thesis and discusses open issues. The appendix introduces mathematical notation (Appendix A), lists the source code of our tool implementation Gemia (Appendix B) and the source code of the two case studies (Appendix C).

1.3. Contributions

Being developed within the research project *Foundations of Heterogeneous Specifications Using State Machines and Temporal Logic* mentioned in Section 1.1, this thesis closes all of Gaps 1–7 and makes a first step towards closing Gap 8. Gap 9 is partially closed by supporting some of the logical operators and showing that some others cannot be supported in interface theories. Several previous interface theories attempted to close some of these gaps, however, the complex interdependencies of the above properties made these attempts sacrifice other desired properties. The particular challenge that has been solved in this thesis is to address all of these properties simultaneously within a single interface theory.

Parts of this dissertation have been published in research papers. These publications and the author's contributions to these are as follows:

- G. Lüttgen, W. Vogler, S. Fendrich. "Richer Interface Automata with Optimistic and Pessimistic Compatibility" In: *Acta Informatica* 52.4–5 (2015), pp. 305–336 [LVF15]: The author contributed an investigation of alphabet extension for optimistic interface theories, which shows that IA-based refinement is insufficient for supporting perspective-based specification because it yields a conjunction that is unintuitive in practice. As a consequence of this result, modal refinement was adopted in later versions of MIA.

- F. Bujtor, S. Fendrich, G. Lüttgen, W. Vogler. "Nondeterministic Modal Interfaces". In: *Theory and Practice of Computer Science (SOFSEM)*. Vol. 8939. LNCS. Springer, 2015, pp. 152–163 [Buj+15], as well as the accompanying technical report [Buj+14]: The author contributed a definition of a quotient operator for MIA, an example of quotienting and a proof that the quotient is adjoint to parallel composition. It is the first quotient for interface theories that supports nondeterministic dividends and compatibility. The author also contributed the—to the best of our knowledge—first discussion of alternative alphabet choices for the quotient. Further, the author adapted conjunction including the relevant proofs to the new refinement preorder introduced as a consequence of [LVF15] (see the first bullet point above).

- F. Bujtor, S. Fendrich, G. Lüttgen, W.Vogler. "Nondeterministic Modal Interfaces". In: *Theoretical Computer Science* 642 (2016), pp. 24–53. [Buj+16]: In addition to the contributions already published in the conference paper [Buj+15] and the technical report [Buj+15], the author discussed in more detail the quotient operator and the difficulties when considering the quotient with regards to nondeterministic divisors and alphabet-extension; these difficulties lead to a non-unique quotient. The author also contributed a more detailed explanation of the conjunction example of [Buj+15] and adapted disjunction to the new refinement preorder introduced as a consequence of [LVF15] (see the first bullet point above). Further, the author conducted a small case study concerning a client-server application (Section 4.2 below) that has been checked with the software tool Gemia presented in Section 4.4.1 below).

- S. Fendrich, G. Lüttgen. "A Generalised Theory of Interface Automata, Component Compatibility and Error". In: *Integrated Formal Methods (iFM)*. Vol. 9681. LNCS. Springer, 2016, pp. 160–175 [FL16b], as well as the accompanying technical report [FL16a]: The author contributed all of the paper's technical content and discussed it extensively with the co-author. This includes the error-preserving interface theory EMIA (Section 3.1 below), which solves Gaps 1–7 listed in Section 1.1 above, and the investigation of error-abstracting and error-preserving interface theories which are related by a Galois insertion. Error-abstraction is identified as a major source of several issues listed in Section 1.1.

The following list comprehensively summarises the author's detailed contributions of this dissertation. A reference to the corresponding section in this thesis is provided and, in

case the contribution is already published, the according scientific publication is cited.

- An abstract theory of specification (Section 2.1): The idea to consider a specification as a description of a set or class of implementations is well-known and often used implicitly, e.g., [Mil71; LX90; FS08; BHW11; Ben+13]. In this thesis we exercise this idea explicitly, yielding a general, algebraic formalisation of the concepts of specifications, composition, refinement, compositionality and adjoint operations. This formalisation permits a precise definition of heterogeneous specification as a combination of domain-specific and logic-based specification. We also relate this formalisation to the concept of independent implementability as employed, e.g., in software engineering. A benefit of this abstract formalisation is that we get a better understanding of the role of logical operations such as conjunction and disjunction and the idea of heterogeneous specifications. Further, we can already prove several properties of adjoint operations at this abstract level such that we get these properties for free when instantiating with concrete operations, e.g., the quotient.

- A comparative study of specification theories (Section 2.4): A comparative study of 27 specification theories for concurrent systems with respect to 21 features is conducted. To our knowledge this is the largest study of that kind.

- A general definition of the concept of alphabet extension (Section 3.1.9): Alphabet extension is used in interface theories since the first paper by de Alfaro and Henzinger on Interface Automata (IA) [AH01a]. Alphabet extension operators are either provided explicitly, e.g., [Buj+16; Rac+11], or implicitly included in the refinement preorder, e.g., [AH01a; LVF15]. However, only concrete alphabet extension operators are considered in the literature. This thesis provides a general definition of the concept of an alphabet extension operator whereof the above operators are concrete instances. Such a formalisation permits one to separate the concept of extension from desired properties, e.g., monotonicity, and eases the study of nonuniform extension operators where different variants of extension are intermixed.

- An investigation of alphabet extension for interface theories (Sections 3.1.9 and 3.4, published in [LVF15]): Different concrete variants of alphabet extension operators that add transition loops to states of an interface are investigated, showing that IA-based refinement is insufficient for perspective-based specification. This insight provides the reason for reconsidering the MIA-theory of [LVF15] under a different refinement in [Buj+16] (the reconsideration is joint work with the co-authors of the latter article).

- A quotient operator for MIA (Sections 3.1.7 and 3.2.2, published in [Buj+15; Buj+16]): As a part of the reconsideration mentioned in the previous contribution, the author contributed a quotient operator that supports incremental design and

component reuse. In particular, this quotient operator is the first one supporting nondeterministic dividends and compatibility. The quotient is shown to be monotonic in the left argument (Lemma 2.1(2)). The difficulties of generalising the quotient to nondeterministic divisors are discussed. Further, it is shown that relaxing the alphabet conditions on the operands renders the quotient nonunique for all interface theories based on IA, a question that has not been considered before.

- The error-preserving interface theory EMIA (Section 3.1, published in [FL16b]): The interface theory EMIA is developed and solves all of Gaps 1–7 listed in Section 1.1 due to a refined semantics based on error-preservation. As a consequence, EMIA is the most general and most practical interface theory to date. In addition to the publication [FL16b], universal states, hiding and restriction are adapted from [Buj+16] and a quotient operator for EMIA is contributed including additional properties of quotienting for MIA and EMIA (see Lemma 2.1), when compared with [Buj+16], where only item (2) of Lemma 2.1 is proved.

- An investigation of error-preserving and error-abstracting interface theories (Sections 3.2 and 3.3, published in [FL16b]): The error-abstraction employed in previous interface theories is shown to be the source of several issues listed in Section 1.1. The mathematical relation between error-abstraction and EMIA's error-preservation is shown to relate MIA and EMIA by a Galois insertion. This results in a better understanding of the role of error-abstraction in interface theories, in particular, because the Galois insertion not only relates interface specifications but also manifests itself at the levels of operations on interfaces and proofs of properties.

- An investigation of implication and negation for interface theories (Section 3.1.8): As a result of the abstract specification theory mentioned above, it is natural to investigate implication as an adjoint of conjunction, and negation as a special case of implication. We show, that any specification theory based on (finite state) MTS is not closed under negation and, therefore, implication and negation cannot be provided in practice. An underapproximation of implication presented in [GR09] is shown to be impractical, too.

- A case study on a client-server application (Section 4.2, published in [Buj+16]): The practical utility of the MIA-theory is demonstrated in the design of a nondeterministic client-server application. In particular, this example shows that, in general, deterministic interface theories are insufficient for software design.

- A case study concerning parts of a railway control system (Section 4.3): The practical utility of the EMIA-theory is demonstrated by applying it as a specification language to the design of a safety-critical railway control system that has to secure a railway junction. The whole case study has been conducted with our software-tool MiaGo that is described below and implemented in Go.

- An implementation of MIA in Haskell (Section 4.4): An embedded DSL providing several operators of the MIA-theory has been implemented in Haskell in order to check many examples including the above client-server case study. In addition, the tool has been used for cross-checking the Go-implementation below.
- An implementation of EMIA in Go (Section 4.4): Based on an existing implementation of MIA [Gar15] that was written by a Master's student supervised by the author, EMIA has been implemented as a software tool in order to demonstrate its practical applicability.
- An investigation of interface theories as behavioural types (Section 5): While interface theories are a well studied family of specification languages for software design, their uniform interaction model makes them unsuitable for transitioning to implementation. Therefore, a typed version of IA called IIIA is developed as a behavioural type theory, and standard properties of type systems such as subject reduction, congruence and type safety are proved. In particular, type checking may be seen as a syntactic approximation of refinement checking that is computationally more efficient. The weaknesses of IA, in contrast to MIA, as a foundation are analysed, and the differences and commonalities to the well-known session types, e.g., [HYC16], are discussed. This proof of concept shows the principle applicability of interface theories as behavioural type theories, and the author believes that this work may start a promising new research direction for interface theories.

2. Preliminaries

This chapter defines some basic concepts related to formal specification and develops a formal theory of specification. A summary of formalisms for specifying concurrent reactive systems is given with a focus on those formalisms that are relevant to the present thesis, in particular, on interface theories.

2.1. A Formal Theory of Specification

In this section we develop an abstract theory of specification that is suitable for comparing different concrete specification theories on a common basis. In contrast to [FS08] and [Bau+12], where specification theories are devised for special purposes, we define a general purpose concept of specification theory that serves a better understanding of the concepts of specification, composition, refinement, compositionality and adjoint operations. This formalisation also permits a precise definition of heterogeneous specification as a combination of domain-specific and logic-based specification. Further, we relate this formalisation to the concept of independent implementability as employed, e.g., in software engineering.

A *problem domain* is given by a collection of problems and a collection of potential solutions to problems. For example, the problem of a task that is to be done may have a computer program completing the task as a solution. A *compositional problem domain* additionally has operations on the solutions, which allow one to compose more complex solutions from simpler ones. In the case of the above computer program, the program might be composed of simpler programs that perform subtasks of the original problem.

For a given problem, one is interested in describing which solutions one considers as admissible. Mathematically, this can be done by giving a set of admissible solutions. Such an *extensional description* renders the powerset over all possible solutions into a specification theory that plays a central role for comparing different specification theories over the same problem domain. However, a set of admissible solutions may be very large, e.g., infinite, such that an extensional representation is impractical. The purpose of a *specification* is to provide a finite *intensional description* of a set of admissible solutions for a given problem. These solutions are called the *implementations* of the specification.

A specification theory should provide the problem domain's composition operations as well as the set-theoretical operations intersection, union and inclusion on the intensional level, i.e., on the level of specifications. Figure 2.1 illustrates this relation between a specification theory $\mathcal{S}$ with a given interpretation function $[\![\cdot]\!]$ into the powerset of a set I of implementations, where $\circ$ is a placeholder for operations of the underlying problem domain and of set theory. Note that, in general, $\circ_{\mathcal{S}}$ only approximates $\circ_{\mathfrak{P}}$ and the square

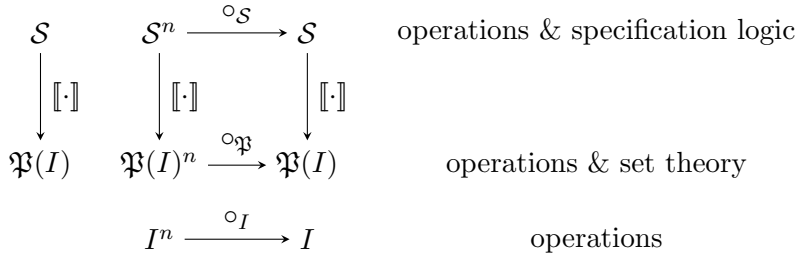


Figure 2.1.: An interpretation $[\![\cdot]\!] \colon \mathcal{S} \to \mathfrak{P}(I)$ of specifications $\mathcal{S}$ as subsets of a universe of implementations I should reflect operations $\circ$ of a problem domain and of set theory.

is not required to commute.

Because the set-theoretical operations turn the powerset $\mathfrak{P}(I)$ into a boolean lattice, it is quite natural to use classical propositional logic or fragments thereof as a basis for a specification formalism. However, depending on the nature of the problem domain, some domain specific formalism may be more suitable for describing certain aspects of a specification. Ideally, one would be able to combine logic based formalisms with domain specific formalisms into a *heterogeneous* specification theory which permits a designer to choose the most suitable description for each aspect of a specification. In such an approach, conjunction is a particularly important operator, which allows one to combine specifications of such aspects into an overall specification as desired for perspective-based specification.

2.1.1. Specification Theories

For the purpose of investigating the concept of specification formally, the problems themselves are not of much interest to us. A specification theory should be agnostic with respect to problems because, in general, the problems may even consist of rather vague ideas for which one cannot give a formal description. Recalling the above example, a systems designer may be able to decide whether a specific computer program solves a given task, even if he or she is unable to formally describe the task in full detail. In such a case, a specification may also serve to make an informal description more precise. Therefore, we intentionally omit the problems from our formal theory and consider only implementations.

Formally, a *compositional problem domain* is given by a pair $(\mathcal{A}, \equiv)$ consisting of a Σ-algebra $\mathcal{A}$ of implementations and a congruence relation $\equiv$ on $\mathcal{A}$. The signature Σ describes which operations are defined on implementations and the congruence relation $\equiv$ abstracts from implementation details that are irrelevant with respect to the admissibility of problem solutions. Implementations related by $\equiv$ are called *semantically equivalent*. In the following we abbreviate $(\mathcal{A}, \equiv)$ by $\mathcal{A}$.

Let $\Sigma := (F_\Sigma, \mathrm{ar}_\Sigma)$ be a signature, $\mathcal{A}$ a problem domain over Σ, and $\Lambda := \{\sqcap, \sqcup, \to\}$ the set of propositional logic operators *conjunction*, *disjunction* and *implication*, re-

spectively, with arities $\mathrm{ar}_\Lambda := \{(\sqcap, 2), (\sqcup, 2), (\rightarrow, 2)\}$. A *specification theory* on $\mathcal{A}$ is a tuple $(\mathcal{S}, \sqsubseteq, L, [\![\cdot]\!])$, where $L \subseteq \Lambda$ and $\mathcal{S}$ is a Σ'-algebra over the signature $\Sigma' := (F_\Sigma \cup L, \mathrm{ar}_\Sigma \cup \mathrm{ar}_L)$, $\sqsubseteq$ is a preorder on $|\mathcal{S}|$ and $[\![\cdot]\!]\colon |\mathcal{S}| \to \mathfrak{P}(|\mathcal{A}|)$ is a function. The elements of $|\mathcal{S}|$ are called the *specifications* of $\mathcal{S}$, $\sqsubseteq$ the *refinement preorder* and $[\![\cdot]\!]$ the *interpretation function* that assigns each specification its set of implementations. The logic operators and the refinement preorder represent set intersection, set union, implication, and set inclusion on the intensional level, respectively.

If $s \in |\mathcal{S}|$ and $i \in [\![s]\!]$, we say that i implements s, which we sometimes write as $i \vDash s$. A specification theory is *sound* with respect to $\equiv$ if, for all $i \in [\![s]\!]$, we have $[i]_\equiv \subseteq [\![s]\!]$. Soundness is an important property because in an unsound specification theory, if i implements s, an equivalent implementation $j \equiv i$ does not necessarily also implement s. We say, a specification theory *reflects* an operation $f \in F_\Sigma$, if $[\![\cdot]\!]$ is homomorphic with respect to $f^{\mathcal{A}}$. Otherwise, it is said to *approximate* f.

While set inclusion provides a mathematically convenient way to compare specifications with respect to the implementations they permit, this comparison is computationally intractable for infinite implementation sets. In order to sufficiently approximate set inclusion, a minimal requirement on a specification theory is that $[\![\cdot]\!]$ is monotonic with respect to the refinement preorder $\sqsubseteq$, i.e., that $s \sqsubseteq s'$ implies $[\![s]\!] \subseteq [\![s']\!]$. Under a monotonic interpretation, refining a specification means restricting its set of implementations, i.e., refining means becoming more specific. In the other direction, if $[\![s]\!] \subseteq [\![s']\!]$ implies $s \sqsubseteq s'$, then $\sqsubseteq$ is said to be *thorough*. Thorough refinement represents the ideal world. However, in many cases thorough refinement is computationally hard and, therefore, one often prefers a non-thorough approximation [Ben+15]. The refinement preorder is called *compositional* with respect to an operation $f \in F_{\Sigma'}$, if $s_1 \sqsubseteq t_1, \ldots, s_{\mathrm{ar}\, f} \sqsubseteq t_{\mathrm{ar}\, f}$ implies $f^{\mathcal{S}}(s_1, \ldots, s_{\mathrm{ar}\, f}) \sqsubseteq f^{\mathcal{S}}(t_1, \ldots, t_{\mathrm{ar}\, f})$, i.e., if $f^{\mathcal{S}}$ is monotonic in each argument. A specification theory is called compositional if its refinement preorder is compositional with respect to all operations. A specification theory is particularly well behaved if conjunction $\sqcap$ and disjunction $\sqcup$ correspond to the greatest lower bound and the least upper bound with respect to $\sqsubseteq$, respectively. In such a case, it is easy to see that $\sqsubseteq$ is also compositional with respect to $\sqcap$ and $\sqcup$.

Compositionality of the refinement preorder and monotonicity of the interpretation enable a systems designer to employ the principle of *independent implementability* introduced in Section 1.1 and illustrated in Figure 2.2. Starting with coarse specifications $S_1, \ldots, S_k$ of the components of a system $f^{\mathcal{S}}(S_1, \ldots, S_k)$, one may refine these specifications stepwise until one reaches an implementation, i.e., one may substitute less specific specifications by more specific specifications $R_1^{(i_1)}, \ldots, R_k^{(i_k)}$ during the design process. Due to the monotonicity of the interpretation, this approach preserves implementation properties of a specification, i.e., if $s \sqsubseteq s'$, then any property that is true for all implementations of s' is obviously true for the implementations of s. With compositionality, one may also apply this approach to components of a system while keeping the desired properties of the system as a whole. A practical application of this process can be found in Chapter 4.

For a binary operation $\circ$ we may also be interested in an inverse operation, i.e., an

	Components			System
Specification	S_1	$\dots$	S_k	$f^{\mathcal{S}}(S_1, \dots, S_k)$
	$\sqcup\!\mid$		$\sqcup\!\mid$	
	$R_1^{(1)}$	$\dots$	$R_k^{(1)}$	$\sqcup\!\mid$
	$\sqcup\!\mid$		$\sqcup\!\mid$	
Refinements	$\vdots$		$\vdots$	$f^{\mathcal{S}}(R_1^{(i_1)}, \dots, R_k^{(i_k)})$
	$\sqcup\!\mid$		$\sqcup\!\mid$	
	$R_1^{(n_1)}$	$\dots$	$R_k^{(n_k)}$	$\sqcup\!\mid$
	$\sqcup\!\mid$		$\sqcup\!\mid$	
Implementation	I_1	$\dots$	I_k	$f^{\mathcal{A}}(I_1, \dots, I_k)$

Figure 2.2.: The design approach of independent implementability.

operation $\triangleright$ such that $t \circ (t \triangleright s) \equiv s$. Of course, such an inverse does not exist for all interesting operations $\circ$. However, it may sometimes still be sufficiently approximated by an operation that is *adjoint* [Mac71] to $\circ$, i.e., that yields a largest specification $t \triangleright s$ satisfying the bound $t \circ (t \triangleright s) \sqsubseteq s$. Hence, an adjoint operation is defined by $x \sqsubseteq t \triangleright s \iff t \circ x \sqsubseteq s$. We can show some general properties of adjoint operations:

Lemma 2.1. *For any specification theory with an operation $\circ$ and an operation $\triangleright$ that is adjoint to $\circ$, we have*

1. $t \sqsubseteq s \triangleright (s \circ t)$,
2. $t \sqsubseteq t' \implies s \triangleright t \sqsubseteq s \triangleright t'$,
3. $s \sqsubseteq s' \implies s \triangleright t \sqsupseteq s' \triangleright t$, *if $\sqsubseteq$ is compositional with respect to $\circ$,*
4. $s \circ (s \triangleright (s \circ t)) \equiv_{\sqsubseteq} s \circ t$, *if $\sqsubseteq$ is compositional with respect to $\circ$,*
5. $(s \sqcup t) \triangleright u \sqsubseteq (s \triangleright u) \sqcap (t \triangleright u)$, *if $\sqsubseteq$ is compositional with respect to $\circ$,*
6. $(s \circ t) \triangleright u \equiv_{\sqsubseteq} t \triangleright (s \triangleright u)$, *if $\circ$ is associative.*

Proof. Claim 1: By definition of adjoints, we have $x \sqsubseteq s \triangleright (s \circ t) \iff s \circ x \sqsubseteq s \circ t$. Due to reflexivity of $\sqsubseteq$, t is a solution for x in the right inequation. Hence, t is also a solution in the left inequation.

Claim 2: By definition of adjoints, $x \sqsubseteq s \triangleright t$ implies $s \circ x \sqsubseteq t$. Applying the assumption $t \sqsubseteq t'$ and the transitivity of $\sqsubseteq$, we conclude that $s \circ x \sqsubseteq t'$. The definition of adjoints implies $x \sqsubseteq s \triangleright t'$. By reflexivity of $\sqsubseteq$ we may substitute $s \triangleright t$ for x.

Claim 3: By definition of adjoints, $x \sqsubseteq s' \triangleright t$ implies $s' \circ x \sqsubseteq t$. Our assumption $s \sqsubseteq s'$ and the compositionality of $\circ$ imply $s \circ x \sqsubseteq s' \circ x$. Transitivity of $\sqsubseteq$ yields $s \circ x \sqsubseteq t$. Then, the definition of adjoints implies $x \sqsubseteq s \triangleright t$. By reflexivity of $\sqsubseteq$, we may substitute $s' \triangleright t$ for x.

Claim 4: Direction "$\sqsubseteq$" follows from $s \triangleright (s \circ t) \sqsubseteq s \triangleright (s \circ t)$ and the definition of adjoint operations. Direction "$\sqsupseteq$" is a direct consequence of Claim 1 and compositionality.

Claim 5: $x \sqsubseteq (s \sqcup t) \triangleright u \iff (s \sqcup t) \circ x \sqsubseteq u \implies (s \circ x) \sqsubseteq u \wedge (t \circ x) \sqsubseteq u \iff x \sqsubseteq s \triangleright u \wedge x \sqsubseteq t \triangleright u \iff x \sqsubseteq (s \triangleright u) \sqcap (t \triangleright u)$.

Claim 6: By applying the definition of adjoint operations, we get the equivalences $x \sqsubseteq (s \circ t) \triangleright u \iff s \circ t \circ x \sqsubseteq u \iff t \circ x \sqsubseteq s \triangleright u \iff x \sqsubseteq t \triangleright (s \triangleright u)$. □

For instance, defining $s \triangleright t := s \implies t$ makes implication the adjoint of logical conjunction $\wedge$ with respect to the entailment preorder. Then, Lemma 2.1(6) instantiates to currying. Similarly, the quotient operator $/\!\!/$ present in many concurrent specification theories is adjoint to parallel composition when defining $s \triangleright t := t /\!\!/ s$ (see Section 3.1.7).

2.1.2. Comparing Specification Theories over a Problem Domain

In this section we define some generic specification theories over arbitrary problem domains. We also discuss how one may compare different specification theories over the same problem domain, in particular, with respect to their expressiveness.

The extensional representation mentioned above can be formalised as a specification theory:

Definition 2.2 (Extensional Specification Theory)**.** Given a problem domain $\mathcal{A}$, the *extensional specification theory* over $\mathcal{A}$ is defined by the carrier set $|\mathcal{S}| := \mathfrak{P}(|\mathcal{A}|/\equiv)$ with the functions $f^{\mathcal{S}}(s_1, \ldots, s_{\mathrm{ar}\, f}) := \{f^{\mathcal{A}}(i_1, \ldots, i_{\mathrm{ar}\, f}) \mid i_1 \vDash s_1, \ldots, i_{\mathrm{ar}\, f} \vDash s_{\mathrm{ar}\, f}\}$, for all $f \in F_\Sigma$, the logical operators $\sqcap := \cap$, $\sqcup := \cup$ and $\to := (s, s') \mapsto \complement s \cup s'$, the refinement preorder $\sqsubseteq := \subseteq$ and the interpretation function $[\![\cdot]\!] : s \mapsto \bigcup s$.

Obviously, the extensional specification theory of a problem domain $\mathcal{A}$ is sound and thorough and reflects all operations. Because the powerset is a boolean algebra, extensional specification theories permit a natural interpretation of implication and negation and, hence, provide a propositional logic of specification. Implication, as an adjoint of conjunction, is defined by $x \sqsubseteq s \to t \iff x \sqcap s \sqsubseteq t$, whereas negation is a special case of implication, where t is the inconsistent specification $\bot$. In an extensional specification theory, this equivalence translates to $x \subseteq s \to t \iff x \cap s \subseteq t$. Hence, $s \to t$ is the largest set, whose intersection with s is contained in t, namely $s \to t = \complement s \cup t$. Therefore, negation results in complementation, $\neg s = \complement s$.

Because the interpretation function $[\![\cdot]\!]$ of a specification theory defines an implementation relation $\vDash$, each specification $\mathcal{S}$ over a problem domain $\mathcal{A}$ defines a formal context $(|\mathcal{A}|, |\mathcal{S}|, \vDash)$, which gives rise to a concept lattice $\mathfrak{B}(|\mathcal{A}|, |\mathcal{S}|, \vDash)$. This concept lattice may also be considered as a specification theory over $\mathcal{A}$:

Definition 2.3 (Conceptual Specification Theory)**.** Let $\mathcal{S}$ be a specification theory over a problem domain $\mathcal{A}$. The *conceptual specification theory* of $\mathcal{S}$ is defined by the carrier set $\mathcal{C} := |\mathfrak{B}(|\mathcal{A}|, |\mathcal{S}|, \vDash)|$, where $\sqcup$, $\sqcap$ and $\sqsubseteq$ are inherited from $\mathfrak{B}(|\mathcal{A}|, |\mathcal{S}|, \vDash)$. That is, each specification is a formal concept $c = (A, S)$, where the sets $A \subseteq |\mathcal{A}|$ and $S \subseteq |\mathcal{S}|$ are mutually closed with respect to $\vDash$. The interpretation function assigns each specification

its extension, i.e., $[\![\cdot]\!] : (A, S) \mapsto A$. Given specifications $c_1, \ldots, c_{\mathrm{ar}\, f} \in \mathcal{C}$, the function $f^{\mathcal{C}}$ maps to the concept generated by the set $I_f(c_1, \ldots, c_{\mathrm{ar}\, f}) := \{f^{\mathcal{A}}(i_1, \ldots, i_{\mathrm{ar}\, f}) \mid i_1 \vDash c_1, \ldots, i_{\mathrm{ar}\, f} \vDash c_{\mathrm{ar}\, f}\}$, i.e., $f^{\mathcal{C}}(c_1, \ldots, c_{\mathrm{ar}\, f}) := (I_f(c_1, \ldots, c_{\mathrm{ar}\, f})'', I_f(c_1, \ldots, c_{\mathrm{ar}\, f})')$.

Obviously, the conceptual specification theory of $\mathcal{S}$ is sound if $\mathcal{S}$ is sound, and it is always thorough and closed under $\sqcap$ and $\sqcup$, even if $\mathcal{S}$ is not.

On a preordered problem domain, intervals may serve as specifications, where larger intervals represent coarser specifications:

Definition 2.4 (Interval Specification Theories)**.** Let $(\mathcal{A}, \lesssim)$ be a preordered problem domain where $f^{\mathcal{A}}$ is monotonic with respect to $\lesssim$ for all $f \in F_\Sigma$. We define the *interval specification theory* of $\mathcal{A}$ by $|\mathcal{S}| := \{(l, u) \in |\mathcal{A}|^2 \mid l \lesssim u\}$, $(l_1, u_1) \sqsubseteq (l_2, u_2)$ iff $l_2 \lesssim l_1$ and $u_1 \lesssim u_2$, and the interpretation function $[\![(l, u)]\!] := \{i \in |\mathcal{A}| \mid l \lesssim i \lesssim u\}$. The functions are defined by $f^{\mathcal{S}}((l_1, u_1), \ldots, (l_{\mathrm{ar}\, f}, u_{\mathrm{ar}\, f})) := (f^{\mathcal{A}}(l_1, \ldots, l_{\mathrm{ar}\, f}), f^{\mathcal{A}}(u_1, \ldots, u_{\mathrm{ar}\, f}))$. Due to the transitivity of $\lesssim$, if $l_1 \equiv_{\lesssim} l_2$, $u_1 \equiv_{\lesssim} u_2$, $l_1 \lesssim u_1$ and $l_2 \lesssim u_2$, then any $i \in |\mathcal{A}|$ satisfies $l_1 \lesssim i \lesssim u_1$ iff $l_2 \lesssim i \lesssim u_2$; hence, $[\![(l_1, u_1)]\!] = [\![(l_2, u_2)]\!]$ and $(l_1, u_1) \equiv_{\sqsubseteq} (l_2, u_2)$.

When investigating specification theories $\mathcal{S}$ and $\mathcal{T}$ over the same problem domain $(\mathcal{A}, \equiv)$, one wishes to compare them with respect to the sets of implications they may express. In this sense, $\mathcal{S}$ is less or equally expressive than $\mathcal{T}$, if $\{[\![s]\!] \mid s \in |\mathcal{S}|\} \subseteq \{[\![t]\!] \mid t \in |\mathcal{T}|\}$.

Obviously, the extensional specification theory is always the most expressive and the empty specification theory the least expressive. The conceptual specification theory of $\mathcal{S}$ is at least as expressive than $\mathcal{S}$ and is the least expressive specification theory that is complete with respect to conjunctions and disjunctions of $\mathcal{S}$. In particular, if $\mathcal{S}$ is complete, then it is equally expressive than its conceptual specification theory.

2.1.3. Related Work

In order to compare their *One-selecting Modal Transition Systems* (1MTS) to *Disjunctive Modal Transition Systems* (DMTS), Fecher and Schmidt [FS08] define implementation settings and abstraction settings. An implementation setting is a set I of implementations together with an equivalence relation $\equiv$ on I. In contrast to our problem domains, operations and compositionality are not considered. An abstraction setting over an implementation setting $(I, \equiv)$ is a preordered set $(A, \sqsubseteq)$ of abstractions together with an embedding $h : I \to A$ satisfying $i_1 \equiv i_2 \iff h(i_1) \sqsubseteq h(i_2)$. This condition requires that mutual refinement coincides with $\equiv$ on implementations and that every implementation is representable by some specification. Such requirements are acceptable for theories based on MTS. However, they render abstraction settings less general than our definition of specification theories.

Bauer et al. [Bau+12] define a concept of specification theory in order to study assume/guarantee contracts. There, a specification theory is a preordered set with a composition operation. Such specifications are not related to implementations but serve themselves as implementations of contracts. Similarly, the composition operator, its adjoint quotient and conjunction are only used to construct specifications from contracts and are not related to implementations and to the fundamental logic of the extensional

specification theory. Our more general framework subsumes this approach by a problem domain over the signature $(\otimes, \mathrm{ar})$, where $\mathrm{ar} \otimes = 2$.

An alternative approach to specification based on category theory and model theory is provided by *institutions* [GB92]. Institutions define an abstract framework for comparing logical systems and are mainly targeted to specification and verification of algebraic data types in functional programming languages. The basic idea is that theories represent specifications and models represent implementations. This is similar to our approach; however, institutions employ a much more complex mathematical machinery because they are intended to capture structural properties of specifications when translating between different specification languages.

2.2. Formalisms for Specifying Concurrent Systems

This section is intended as a summary of specification languages for concurrent systems. Due to the many existing languages we restrict ourselves to a selection of well-known languages.

2.2.1. Overview of Specification Languages for Concurrent Systems

There are several description languages that are widely used in practice, such as the *Specification and Description Language* (SDL) [Sdl17], behavioural diagrams of the *Unified Modeling Language* (UML) [Uml17], *Event-driven Process Chains* (EPCs) [KNS92], *Business Process Model and Notation* (BPMN) [Bpmn17] and *Business Process Execution Language* (BPEL) [Bpel17]. These languages do not provide a formal semantics and, therefore, may rather be seen as a syntactic front-end to specification than as a foundation of system verification and design.

Two well-known refinement-based approaches to systems design are the *Z notation* [Spi92] and the *B-method* [Abr+91]. The Z notation employs predicate logic for describing the effect of system operations. Modularisation is achieved by employing so-called schema which collect descriptions of state spaces, pre- and post-conditions, invariants, state changes and relationships between abstract and concrete specifications. While this enables a refinement-based approach, one has to manually prove that invariants are maintained. The B-method employs a syntactically different but similar approach with a stronger focus on generating proof obligations, e.g., via weakest preconditions, that have to be shown in order to maintain invariants. In contrast to the Z notation, the B-method explicitly supports concurrency by means of a parallel operator. In both approaches, refinement is targeted towards *data refinement* and *algorithmic refinement*, i.e., the concretisation of data structures and algorithms from abstract behaviours, in contrast to the behavioural refinement employed in interface theories. Both the Z notation and the B-method have been combined with process algebras (Section 2.2.2) such as CSP [Hoa85] in order to address concurrency, e.g., the Circus language [WC01] combines CSP with Z, whereas [But00] presents a combination of CSP and B.

The remaining languages presented here may serve as models of concurrency. Following [WN93] we distinguish interleaving and noninterleaving models. Examples of noninterleaving models are Petri nets [Pet62], event structures [Win87] and Mazurkiewicz traces [Maz88]. Their fundamental view on concurrency is that actions may be independent and, thus, the components of a system may perform their actions simultaneously. Noninterleaving models support a refinement notion called *action refinement*, where an action may be replaced by a more complex behaviour, e.g., [GR01]. In contrast, interleaving models, such as transition systems or trace models, assume that simultaneous actions are not actually happening at the same time but nondeterministically interleaved in an arbitrary order. A more detailed comparison may be found in [Gla01; Gla93; WN93]. The usual refinement notion employed in interleaving models is *behavioural refinement* such as trace inclusion, failures, testing or various forms of simulation (see [BPS01]).

Further, we may distinguish synchronous and asynchronous concurrency. In *synchronous concurrency*, parallel processes synchronise their input and output channels on a regular basis defined by a clock tick. Languages supporting synchronous concurrency are typically employed in systems with real-time requirements, such as hardware circuits, signal processing or controllers of physical systems. *Asynchronous concurrency* considers processes as independent entities that synchronise via communication, e.g., by sending messages or by performing actions that are observable by other processes. Typical applications of asynchronous concurrency are found in distributed, service-oriented software systems.

We also distinguish *synchronous communication*, where a communication signal is immediate, from *asynchronous communication*, where messages may be delayed, e.g., by employing message queues.

The work investigated in this thesis employs an interleaving model of asynchronous concurrency with synchronous communication. It is based on *labelled transition systems* with a *bisimulation semantics* [Mil80] within a *simulation-based* refinement preorder [Gla01; Gla93].

2.2.2. Process Algebras and Labelled Transition Systems

Concurrent systems may be represented algebraically by means of process algebras such as CCS [Mil80; Mil89], ACP [BK82], CSP [Hoa85] or the π-calculus [MPW92]. See [BPS01] for a more thorough treatment of the topic.

Process algebras permit one to compose processes from simpler ones via algebraic operations. Basically, a process may engage in an action a out of an action alphabet A or deadlock/terminate which is denoted 0. More complex processes may be constructed by the sequencing operator . (dot), i.e., $a.0$ denotes a process that first engages in action a and then terminates. The branching operator + permits a process to choose between alternatives, i.e., $p+q$ specifies a process that may choose between behaving like process p or like process q. For instance, a process $a.0 + b.0$ may choose between engaging in action a or b, and afterwards terminates in either case. Concurrency is expressed via the parallel operator $\|$, where $p \| q$ denotes a process that runs processes p and q in parallel. The possible actions are usually sending and receiving messages or signals on channels

that are given by channel names. Parallel processes may interleave or synchronise on their actions. A term $\mu X.\, p(X)$, where X occurs freely in process term p, expresses recursion via the fixed point operator μ.

Many variants of process algebras have been presented in the literature, which differ in details such as available actions, interleaving and synchronisation of actions, or treatment of internal behaviour, i.e., actions that are invisible to other processes. Some process algebras, in particular those based on the π-calculus, support higher-order features such as channel creation and name passing (sending and receiving channel names). A more abstract view on different variants of the π-calculus may be gained through ψ-calculi [Ben+11b].

Central questions of process algebras are behavioural relations and congruences. That is, one is interested whether a process term p may substitute a process term q, e.g., because p exhibits only behaviour that may also be exhibited by q. A rigorous study of behavioural equivalences may be found in [Gla01; Gla93] on the basis of labelled transition systems.

Labelled transition systems (LTS) provide a semantic foundation of process algebras [BPS01] and may also be used as a graphical specification formalism. Because LTSs will serve us as implementations of our problem domain, we give a formal definition:

Definition 2.5 (Labelled Transition System)**.** A *labelled transition system* (LTS) is a triple $\mathcal{L} := (S, A, \longrightarrow)$, where S is a set of *states*, A is a set called the *action alphabet*, and $\longrightarrow \subseteq S \times A \times S$ is a relation called the *transition relation* of $\mathcal{L}$.

We also define the behavioural relations of similarity and bisimilarity because our semantic model is based on them:

Definition 2.6 (Simulation [Par81])**.** Let $\mathcal{L} := (L, A, \longrightarrow_L)$ and $\mathcal{M} := (M, A, \longrightarrow_M)$ be labelled transition systems. A relation $R \subseteq L \times M$ is a *simulation* if for all $(l, m) \in R$,

SI1. $l \xrightarrow{a}_L l'$ implies $\exists m'.\, m \xrightarrow{a}_M m'$ and $(l', m') \in R$.

Two states l and m are called *similar* if there is a simulation R with $(l, m) \in R$.

Definition 2.7 (Bisimulation [Par81])**.** Let $\mathcal{L} := (L, A, \longrightarrow_L)$ and $\mathcal{M} := (M, A, \longrightarrow_M)$ be labelled transition systems. A relation $R \subseteq L \times M$ is a *bisimulation* if for all $(l, m) \in R$,

BI1. $l \xrightarrow{a}_L l'$ implies $\exists m'.\, m \xrightarrow{a}_M m'$ and $(l', m') \in R$,
BI2. $m \xrightarrow{a}_M m'$ implies $\exists l'.\, l \xrightarrow{a}_L l'$ and $(l', m') \in R$.

Two states l and m are called *bisimilar* if there is a bisimulation R with $(l, m) \in R$.

Note that the behavioural equivalence of mutual similarity is coarser than the behavioural equivalence of bisimilarity because for mutual simulation it is sufficient to have two simulation relations, while bisimilarity requires that one relation simulates in both directions simultaneously.

Lüttgen and Vogler extended labelled transition systems into a heterogeneous specification theory called *logic labelled transition systems* (LLTS) [LV06; LV07; LV10] by

providing a conjunction operator and an interpretation of temporal logic operators as labelled transition systems. These ideas have further been developed into the interface theory *Modal Interface Automata* (MIA) [LV12; LV13a; LV13b], which started the line of research to which this thesis belongs (see also Section 2.3).

2.2.3. Modal Transition Systems

Modal Transition Systems (MTS) [LT88] is a family of specification formalisms based on labelled transition systems. MTS supports *underspecification* by means of must- and may-modalities on the transition level. Must-transitions represent transitions that are required, while may-transitions denote permitted transitions.

Definition 2.8 (Modal Transition System [LT88])**.** A *modal transition system* (MTS) is a tuple $\mathcal{M} := (S, A, \longrightarrow, \dashrightarrow)$, where S is a set of *states*, A is a set called the *action alphabet*, $\longrightarrow \subseteq S \times A \times S$ and $\dashrightarrow \subseteq S \times A \times S$ are relations called the *must transition relation* and the *may transition relation* of $\mathcal{M}$, respectively.

An MTS is *syntactically consistent* if and only if every required transition is permitted, i.e., if $\longrightarrow \subseteq \dashrightarrow$.

The idea of underspecification is reflected in the modal refinement preorder. Intuitively, an MTS $\mathcal{L}$ refines an MTS $\mathcal{M}$ if the must-transitions of $\mathcal{M}$ are implemented in $\mathcal{L}$ and the may-transitions of $\mathcal{L}$ are permitted by $\mathcal{M}$.

Definition 2.9 (Modal Refinement [Lar89])**.** Let $\mathcal{L} := (L, A, \longrightarrow_L, \dashrightarrow_L)$ and $\mathcal{M} := (M, A, \longrightarrow_M, \dashrightarrow_M)$ be modal transition systems. A relation $R \subseteq L \times M$ is a *modal refinement relation* if for all $(l, m) \in R$,

MTS1. $m \xrightarrow{a}_M m'$ implies $\exists l'.\, l \xrightarrow{a}_L l' \wedge (l', m') \in R$,
MTS2. $l \overset{a}{\dashrightarrow}_L l'$ implies $\exists m'.\, m \overset{a}{\dashrightarrow}_M m' \wedge (l', m') \in R$.

We say that l refines m and write $l \sqsubseteq_{\mathsf{mts}} m$ if and only if there is a modal refinement relation R between $\mathcal{L}$ and $\mathcal{M}$ with $(l, m) \in R$.

In the following, we omit the indices L and M of the transition relations if the transition systems are clear from the context.

MTS comprises LTS as a special case where $\longrightarrow = \dashrightarrow$. This justifies us to consider LTSs as implementations in the sense that they are syntactically consistent and fully specified, i.e., not subject to underspecification. For LTSs, modal refinement reduces to bisimilarity. Hence, MTS is a specification theory over the problem domain of LTSs and bisimulation equivalence.

When considering the must- and may-transition relations as separate LTSs on the same state set, the must-LTS constitutes a lower and the may-LTS an upper bound on the implementation. However, due to being defined on the same state set, modal refinement is finer than the interval specification theory (Definition 2.4) over LTS with simulation as refinement, because interval refinement reduces to mutual simulation on LTSs.

MTS employs a composition concept which we call *unanimous composition*. A transition is permitted (required) in the composed system, if it is permitted (required) in all

components, i.e., all components of a system unanimously agree on the behaviour of the composition. This is in contrast to the error-aware composition employed in interface theories (Section 2.3).

Many extensions to the basic MTS-formalism have been presented in the literature, such as DMTS [LX90], dMTS [LV13a], 1MTS [FS08] and PMTS [Ben+15], to name a few. *Disjunctive Modal Transition Systems* (DMTS) have been introduced in [LX90] in order to describe the solution sets of equation systems on processes. The variability provided by MTS is insufficient for this purpose because may-transitions can only express optional alternatives. In order to express the requirement that at least one of several alternatives must be implemented, DMTS extends MTS by *disjunctive must-transitions*. That is, the must-transition relation of DMTS is defined as $\longrightarrow \subseteq S \times \mathfrak{P}(A \times S)$. A disjunctive must-transition $s \longrightarrow S'$ with $S' \subseteq A \times S$ expresses the requirement that at least one of the alternatives $(a, s') \in S'$ has to be implemented as a transition $s \xrightarrow{a} s'$. This meaning is reflected in the definition of modal refinement, where Rule MTS1 is replaced by DMTS1:

DMTS1. $m \longrightarrow M'$ implies $\exists L'.\, l \longrightarrow L' \wedge \forall (a, l') \in L'.\, \exists (a, m') \in M'.\, (l', m') \in R$.

A DMTS-specification of a state s may be considered as a description of the required and permitted successor states of s in conjunctive normal form.

The theory *disjunctive Modal Transition Systems* (dMTS)—note the lower case 'd'—has been presented as a foundation of the interface theory MIA in [LV13a]. dMTS is a variant of DMTS that restricts disjunctive must-transitions to single actions and extends DMTS by internal behaviour. Hence, the disjunctive must-transition relation of dMTS is defined as $\longrightarrow \subseteq S \times (A \cup \{\tau\}) \times \mathfrak{P}(S)$, where τ denotes invisible actions. In order to abstract from internal behaviour, *weak transition relations* $\rightarrowtail\!\!\!\!\!\rightarrow$ and $\rightarrowtail\!\!\!\!\!\!\rightarrow$ are defined. Intuitively, a weak transition $\overset{a}{\rightarrowtail\!\!\!\!\!\rightarrow}$ denotes an a-transition that is preceded and succeeded by arbitrarily many τ-transitions. See Definition 3.3 in Section 3.1 for a formal definition. In the definition of modal refinement, Rules MTS1 and MTS2 are replaced by dMTS1 and dMTS2, respectively:

dMTS1. $n \xrightarrow{a} N'$ implies $\exists M'.\, m \overset{a}{\rightarrowtail\!\!\!\!\!\rightarrow} M' \wedge \forall m' \in M'.\, \exists n' \in N'.\, (m', n') \in R$.
dMTS2. $m \dashrightarrow m'$ implies $\exists n'.\, n \overset{a}{\rightarrowtail\!\!\!\!\!\!\rightarrow} n' \wedge (m', n') \in R$.

The restriction to a single action per transition is sufficient for defining conjunction for nondeterministic specifications while avoiding the complications of how to define weak transitions in DMTS, where τ might appear together with visible actions in the same disjunctive must-transition (see Section 3.1). Considering dMTS as a conjunctive normal form as above for DMTS, each clause may only address a single action.

One-selecting Modal Transition Systems (1MTS) [FS08] is a syntactic variant of DMTS that supports disjunctive must-transitions with exclusive choice, i.e., where an implementation must select exactly one of the disjunctive branches of a transition. This feature does not add expressiveness to the theory but permits a more compact representation of some specifications.

Parametric Modal Transition Systems (PMTS) [Ben+15] extend DMTS by global parameters. Each state of a PMTS is specified with obligations in the form of logical formulas that may range over the global parameters and the specified transitions. The global parameters permit one to specify persistent choices, e.g., when choosing between options of a disjunctive transition during refinement of a state s, all states refining s must make the same choice. A deterministic variant which we call PMTS' is also studied, leading to a computationally more efficient check of thorough refinement.

2.2.4. Acceptance Automata

Due to their similarity to DMTS, we summarise *Acceptance Automata* only briefly. Acceptance Automata (AA) have been introduced in [AP91] as a generalisation of *Acceptance Trees* [Hen85]. They have been employed in *Modal Specifications* (MS) [Rac08], a trace-based deterministic variant of MTS, in order to extend the expressiveness of MS to include disjunctive transitions. AA have been used in [Ben+13] in order to define a parallel composition operator for DMTS and as a bridge between νHML (Section 2.2.5) and DMTS, showing that νHML, AA and DMTS are equally expressive.

Definition 2.10 (Acceptance Automaton)**.** An *acceptance automaton* (AA) is a tuple $\mathcal{A} := (S, \mathrm{Acc})$, where S is a set of states and $\mathrm{Acc}\colon S \to \mathfrak{P}(\mathfrak{P}(A \times S))$ is a function that assigns each state a collection of acceptance sets.

The set $\mathrm{Acc}(s)$ represents the permitted implementations of a state s, i.e., an implementation of s chooses one set $S' \in \mathrm{Acc}(s)$ and implements exactly the transitions $s \xrightarrow{a} s'$ given by the elements $(a, s') \in S'$. Hence, an AA-specification of s corresponds to a disjunctive normal form of the required and permitted successor states of s.

Similar to DMTS, difficulties arise with AA when extending it to reasoning about internal behaviour. If a process p comprises an internal loop, then p may have an infinite number of acceptance sets. That is, AA is "too extensional" to allow for a computationally tractable description in the presence of internal behaviour.

2.2.5. Hennessy-Milner Logic and the Modal μ-Calculus

Hennessy-Milner Logic (HML) [HM80] and the *Modal μ-Calculus* ($L\mu$) [Koz83] are two dynamic logics that may be employed for expressing properties of labelled transition systems. The syntax of $L\mu$ is as follows:

$$\begin{aligned}\Phi \quad ::= \quad & p \quad | \quad \top \quad | \quad \bot \quad | \quad \Phi \wedge \Phi \quad | \quad \Phi \vee \Phi \quad | \quad [a]\Phi \quad | \quad \langle a \rangle \Phi \quad | \\ & \neg \Phi \quad | \quad X \quad | \quad \mu X.\, \Phi \quad | \quad \nu X.\, \Phi\end{aligned}$$

p denotes propositonal constants that are interpreted individually for each state of an LTS. The symbols $\top$, $\bot$, $\wedge$, $\vee$ and $\neg$ have the standard meaning of propositional logic. Given an action a, $[a]\Phi$ means after a necessarily Φ, while $\langle a \rangle \Phi$ means after a possibly Φ. The operators μ and ν denote least and greatest fixed points, respectively.

Basically, HML is a fragment of $L\mu$ without the fixed point operators μ and ν. The literature varies about whether p and $\neg$ are included in HML. The fragment νHML that includes only greatest fixed points is particularly interesting because it is equally expressive than DMTS and AA (see Section 2.2.3, Section 2.2.4 and [Ben+13]).

2.2.6. Coalgebras and Interaction Categories

Coalgebras provide a category-theoretic perspective on systems [Rut00]. Therefore, they are very useful for understanding the general mathematical structure of a specification theory. Bisimulation is characterised as a system, i.e., a coalgebra, with a universal property. However, the usual coalgebraic definition of homomorphism is too strong for a characterisation of simulation. Some relaxations of the homomorphism concept, e.g., *lax homomorphisms* [CGH01] and *weak homomorphisms* [CS08], have been proposed in the literature, see also [HJ04].

Given a category $\mathcal{C}$ and an endofunctor $F\colon \mathcal{C} \to \mathcal{C}$, an *$F$-coalgebra* is a $\mathcal{C}$-morphism $c : C \to FC$, for some object $C \in \mathcal{C}$. Coalgebras generalise labelled transition systems to the categorical setting, where the object C represents the state space, the functor F describes the type of labellings, and the morphism c corresponds to the transition relation. For example, a labelled transition system over a set A of action labels is an F-coalgebra over the functor $F\colon \mathbf{Set} \to \mathbf{Set}\colon X \mapsto \mathfrak{P}(A \times X)$.

Although coalgebras are mathematically compelling and a helpful tool for getting a deeper theoretical understanding of systems, we believe that the language of coalgebra is too abstract from a perspective of tool implementation and systems engineering. Therefore, the coalgebraic view is not considered in this thesis.

A different categorical approach to concurrency has been presented as *interaction categories* in [AGN96], with the goal of establishing a Curry-Howard-Lambek-correspondence in the spirit of "processes as morphisms". While focusing on synchronous concurrency, the paper also shows that several promises, such as composition of processes corresponding to composition of morphisms, cannot be kept in the asynchronous case. Hence, interaction categories are not suitable for our asynchronous setting.

2.2.7. Session Types

The specification formalisms presented in the previous subsections employ a uniform communication model, i.e., actions are abstract entities with no internal structure. For example, when action a represents sending a message on channel a, we cannot distinguish whether the sent message is of type integer or type string. In contrast, *session types* are a family of behavioural type systems going back to [Hon93], which may be employed to extend modelling languages for concurrent systems, such as the π-calculus, by typed actions. The main idea behind session types is to define a global type describing the interaction protocol of a communication session. This global type is projected on the communicating components, yielding a local session type for each component. Such a local session type describes the intended behaviour of the component within a session. The type system guarantees that, if each component typechecks against its local session

type, then the whole system adheres to the global session type. Due to the projection on local types, the type check is computationally efficient. A more detailed summary of session types may be found in Chapter 5, where we develop a behavioural type theory similar to session types but based on *Interface Automata* (Section 2.3.2).

2.3. Interface Theories

The specification formalisms presented in the previous subsections provide component models that may be employed in component-based verification. In contrast, *interface theories* [AH01a; AH05; Bau+10; Buj+16; BV14; Che+12; FL16b; LNW07; LVF15; Rac+11], have been developed as specification formalisms for component-based design [AH05; Bau+10; Cai11; KS13] and may serve as specification languages for software product lines [LNW07; FL16b], web services [Bey+07], the Internet of Things [LL15] and conformance testing [LML15; LML17]. Interface theories may also be employed as contract languages or behavioural type theories when transitioning from software design to implementation (see Chapter 5 and [Bau+12; Gar15]).

Interface theories allow one to model the interactive behaviour of a concurrent system's components at the level of their interfaces. One of their main goals is to detect incompatibilities between the communication requirements of different components early in the design phase. To this end, an interface theory has a notion of compatibility as one of its central concepts. This section gives a brief summary of interface theories with a focus on stateful interfaces based on Interface Automata [AH01a].

2.3.1. Assume-Guarantee Interfaces

Assume-guarantee interfaces [AH01b; AH05; Ben+12] is a class of stateless interface theories. Here, an interface is specified by a set of assumptions on input variables and a set of guarantees about output variables, e.g., as logical formulas. If the environment satisfies an interface's input assumptions, then the interface assures its output guarantees. Interfaces P and Q are considered compatible if P's output guarantees satisfy Q's input assumptions and Q's output guarantees satisfy P's input assumptions. Due to their stateless nature, assume-guarantee interfaces are not suited as a model of behaviour and concurrency.

2.3.2. Interface Automata

Many stateful interface theories [LNW07; Bau+10; Rac+11; LVF15; Buj+16] extend de Alfaro and Henzinger's *Interface Automata* (IA) [AH01a; AH05], which are based on labelled transition systems. IA allows one to specify requirements of an interface on its environment by dividing each interface's alphabet of transition labels into input actions ('?'), output actions ('!') and an internal action τ.

Definition 2.11 (Interface Automaton)**.** An *Interface Automaton* (IA) is a tuple $\mathcal{P} := (S, I, O, \longrightarrow)$, where S is a set of *states*, I and O are disjoint sets of *input actions* and

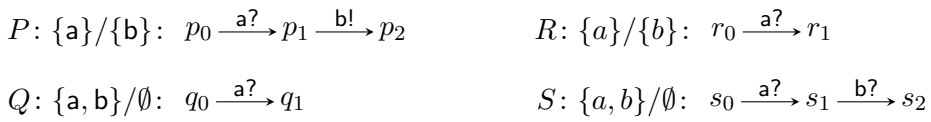


Figure 2.3.: Example of Interface Automata and refinement.

output actions, respectively, the set $A := I \cup O$ is called the *action alphabet* not including the *silent action* τ, and $\longrightarrow \subseteq S \times (A \cup \{\tau\}) \times S$ is a relation called the *transition relation* of $\mathcal{P}$. We also write I/O instead of A when highlighting the input and output alphabets.

Examples of interface automata are given in Figure 2.3. When composing interface automata in parallel, they are assumed to synchronise on shared actions, e.g., action a? in a composition of P and Q. The a?-transition from some state p_0 expresses P's readiness to participate in action a initiated by P's environment. The b!-transition at p_1 expresses that P may initiate b and P's environment must be ready to receive b, i.e., the environment must specify a b?-transition at its current state such as for state s_1 when P is running in environment S. Otherwise, e.g., when employing Q as an environment that does not specify a b?-transition at state q_1, we have a *communication mismatch* and the parallely composed state $p_1 \parallel q_1$ is considered illegal. In this sense, IA-based interface theories are *error-aware* in contrast to the unanimous composition concept of MTS.

Depending on the reachability of illegal states, we distinguish optimistic and pessimistic compatibility. With *optimistic compatibility*, a composed system $P \parallel Q$ is assumed to operate in a helpful environment that tries to steer away from communication mismatches by controlling $P \parallel Q$ via its input transitions. Hence, components P and Q are optimistically compatible if an environment exists in which $P \parallel Q$ cannot reach illegal states. This corresponds to an *open systems view*, where a composition is considered as a part of a larger system, the remaining part of which is responsible to avoid illegal states. In our example from Figure 2.3, P and Q are optimistically compatible because an environment can avoid reaching the illegal state $p_1 \parallel q_1$ by never engaging in output a!. In contrast, *pessimistic compatibility* makes no assumption on the behaviour of the environment of $P \parallel Q$, i.e., $P \parallel Q$ must expect arbitrary behaviour of its environment. Hence, P and Q are pessimistically compatible if illegal states are unreachable in all environments. This corresponds to a *closed systems view*, where a composition is seen as a system by itself and must avoid illegal states autonomously, because an environment exists in which illegal states may be reached. In our example, P and Q are pessimistically incompatible because in an environment that engages in action a as an output the illegal state $p_1 \parallel q_1$ would be reached, whereas P and S are pessimistically compatible. Note that pessimistic compatibility is a stricter concept, i.e., the pessimistically compatible compositions are a subset of the optimistically compatible ones.

Because the larger context may still be in development when composing components of a system during design, IA employs an *open systems view*, i.e., it adheres to optimistic compatibility. When composing components P and Q, one wishes to synthesise the weakest requirement on their environment that guarantees that communication mismatches are avoided. Therefore, the IA parallel composition employs the following pruning of

illegal states. A composed state $p' \parallel q'$ from which an illegal state is reachable by output or internal transitions is also considered illegal, because the environment cannot prevent the system from reaching a mismatch. All illegal states are removed from the parallel composition. This also removes input transitions $p \parallel q \xrightarrow{\mathsf{a}?} p' \parallel q'$ leading to illegal states, including all a-input transitions starting at the same state $p \parallel q$ [BV14]. In the example of Figure 2.3, $P \parallel Q$ consists of a single state $p_0 \parallel q_0$ and no transitions, due to the removal of the illegal state $p_1 \parallel q_1$.

In order to preserve compatibility when refining interfaces, one should not introduce new communication mismatches. A new communication mismatch might be introduced during refinement or implementation by adding a previously unspecified output transition or by removing a previously specified input transition. For this reason, IA considers unspecified outputs as forbidden and specified inputs as mandatory. Vice versa, an existing communication mismatch can be reconciled in refined versions of the components, either by removing the responsible output transition or by adding a suitable input transition with some arbitrary subsequent behaviour. Hence, IA considers specified output transitions as optional and unspecified inputs as permitted with arbitrary subsequent behaviour. This reasoning leads to the definition of alternating refinement, where the weak transition relations $\rightarrowtail\!\!\!\!\!\rightarrow$ and $\longrightarrow\!\!\!\!\!\rightarrow$ abstract from internal behaviour. Intuitively, a weak transition $\overset{a}{\rightarrowtail\!\!\!\!\!\rightarrow}$ denotes an a-transition that is preceded and succeeded by arbitrarily many τ-transitions and a trailing-weak a-transition $\overset{a}{\longrightarrow\!\!\!\!\!\rightarrow}$ denotes one that is only succeeded but not preceded by arbitrarily many τ-transitions. See Definition 3.3 in Section 3.1 for a formal definition.

Definition 2.12 (Alternating Refinement)**.** Let $(L, I, O, \longrightarrow_L)$ and $(M, I, O, \longrightarrow_M)$ be interface automata. A relation $R \subseteq L \times M$ is an *alternating refinement relation* if for all $(l, m) \in R$, $\omega \in O \cup \{\tau\}$ and $i \in I$:

AR1. $l \xrightarrow{\omega}_L l'$ implies $\exists m'.\, m \overset{\omega}{\rightarrowtail\!\!\!\!\!\rightarrow}_M m'$ and $(l', m') \in R$,
AR2. $m \xrightarrow{i}_M m'$ implies $\exists l'.\, l \overset{i}{\longrightarrow\!\!\!\!\!\rightarrow}_L l'$ and $(l', m') \in R$.

Revisiting Figure 2.3, R refines P because we may remove the b-output transition at state p_1, and S refines Q because the b-input transition may be added at state s_1. Note that both refinements resolve the communication mismatch between P and Q.

Under the requirement that an error-free specification may only have error-free implementations, the pruning of parallel composition is theoretically justified in the sense that it renders IA-refinement the coarsest compositional refinement preorder satisfying the requirement [BV14]. Therefore, many IA-based interface theories [LNW07; BV14; Rac+11; LVF15; Buj+16] employ a similar pruning in their definition of parallel composition (Section 2.3.3). However, this thesis also shows that the pruning causes several issues that are undesired in practice (Section 3.3).

2.3.3. Modal Interface Theories

By declaring all output transitions as optional, IA permits designers to remove all outputs and, thus, all functionality, because any actual behaviour is initiated by output transitions.

As a consequence, any interface automaton has a trivial implementation with no actual behaviour. That is, it is impossible to specify required behaviour in IA. This problem has been addressed in IOMTS [LNW07], MIO [Bau+10], MI [Rac+11] and MIA [Buj+15; LVF15] by combining IA with Modal Transition Systems (MTS) [Lar89] or Disjunctive Modal Transition Systems (DMTS) [LX90], which allow one to specify required, optional and forbidden behaviour. Taking stepwise decisions on the optional behaviour allows for an incremental, component-based design that is supported by a compositional refinement preorder based on modal refinement. By combining IA and (D)MTS, modal interface theories promote two main features: requirements on environments and requirements on implementations.

Because unspecified inputs are allowed in IA but forbidden in (D)MTS, several views on how to merge these differences appropriately have been studied in the abovementioned literature. We summarise those results in the following.

In IOMTS [LNW07], this difference is not considered. As a consequence, modal refinement is not compositional with respect to parallel composition as was pointed out in [Rac+11] and proved in [LV12]. In addition, internal must-transitions are not treated properly in IOMTS-refinement [LV13b].

Internal behaviour and the abstraction thereof has been studied in several strong and weak variants of refinement and compatibility in MIO [Bau+10], which is a pessimistic version of IOMTS. The above compositionality issue is irrelevant in MIO because the problematic compositions are undefined in a pessimistic setting.

An attempt to repair the compositionality bug in an optimistic setting has been made in MI [Rac+11] for deterministic interfaces without internal behaviour. When illegal states are removed, MI replaces them by an ordinary state that permits arbitrary behaviour by specifying may-loops for all actions. Although this resolves the compositionality issue, the resulting parallel operator is not associative as shown by Bujtor and Vogler in [Buj+16].

This thesis belongs to the line of research on *Modal Interface Automata* (MIA) [LV13a; LV13b; LVF15; Buj+15; Buj+16; FL16b] that was started by Lüttgen and Vogler with the goal of establishing a nondeterministic interface theory supporting heterogeneous specification. In this thesis we distinguish different variants of MIA by employing the acronyms MIA1 [LV13a], MIA2 [LV13b] and MIA3 [Buj+15; Buj+16]. We append a lower-case letter ‘o’ or ‘p’ to these acronyms, e.g., MIA2o, if we want to highlight that we consider optimistic or pessimistic compatibility, respectively.

MIA1 [LV13a] is the first nondeterministic interface theory that supports conjunction and disjunction with respect to a weak refinement preorder. This preorder is similar to IA-refinement in that specified inputs are mandatory and unspecified inputs are implicitly permitted. For outputs, the full range of MTS-modalities is available. In MIA2 [LV13b] the refinement preorder is modified to properly treat internal must-transitions. Further, a pessimistic version MIA2p is studied including conjunction, disjunction and alphabet extension.

This is where the work on this thesis started. An investigation of alphabet extension for the optimistic setting MIA2o showed that conjunction in MIA2—although mathematically correct—does not yield results as desired in practice, which is a consequence of the

refinement preorder implicitly permitting unspecified inputs like in IA (see Section 3.1.9 and [LVF15]). This led to the adoption of modal refinement in MIA3 [Buj+15; Buj+16], which is the first interface theory that supports optimistic compatibility, nondeterminism, a compositional refinement preorder that abstracts from internal behaviour, hiding, restriction, an associative parallel composition, conjunction, disjunction, alphabet extension and a quotient operator, where alphabet extension and the quotient are contributions by the author of this thesis. Compositionality is achieved by employing a universal state $\top$, similar to MI. However, in contrast to MI's ordinary universal state, $\top$ is treated specially in the definition of refinement and parallel composition, which resolves the associativity issue of MI. Intuitively, state $\top$ allows one to selectively adopt the IA view where necessary. In addition, the work of [LV10] on temporal logic operators has been adapted to MIA3 [BV16].

Although all of the above interface theories are error-aware in the tradition of IA, they are *error-abstracting* in the sense that they consider erroneous behaviour equivalent and eliminate it from the parallel composition, either by considering the parallel composition as undefined (MIO, MIA2p) or by pruning (IA, IOMTS, MI, MIA1, MIA2o, MIA3). As a consequence, interface theories combining IA and MTS have several issues that impact their practical use:

I1. The possibility to redefine unwanted behaviour to not being an error when refining a composition impacts the usability of interface theories for safety-critical systems;
I2. The inability to check the compatibility of multiple components requires one to resort to separate assembly theories [HK15];
I3. A compatibility concept that is too strict for modelling product line families [LNW07];
I4. A global composition concept that requires a fixed view on composition.

These issues led to the development of the *error-preserving* interface theory EMIA [FL16b], which is the main contribution of this thesis (Chapter 3).

2.4. Comparative Study of Specification Theories

Table 2.1 summarises the features provided by different specification theories. We only include theories based on or related to MTS. Each line of the table summarises one theory, and each column a feature. Because some of the theories exist in different versions, we cite a concrete reference. The entries of the table read as follows:

+ The theory fully supports the feature; a proof may be found in the literature or is easy to see.

$\sim$ The feature is partially supported by the theory; the reason why it is not fully supported is obvious in most cases.

− The feature is not supported by the theory; a proof may be found in the literature or is easy to see.

? The feature is not considered in the given paper, however, there is no obvious reason why the theory would not support it.

Table 2.1.: Comparison of MTS- and/or IA-based specification theories with respect to supported features

Theory	Ref.	N	D		τ	P	Par			Cp			And		Or		Quo		θ	α	Asm	Err
			d	D			$\parallel$	$\sqsubseteq$	a	p	o	u	$\sqcap$	glb	$\sqcup$	lub	$/\!\!/$	cp				
MTS	[LT88]	+	−	−	−	−	+	+	+	−	−	+	−	n.a.	?	n.a.	?	n.a.	−	?	n.a.	−
MS	[Rac08]	−	−	−	−	−	+	+	+	−	−	+	+	+	+	+	+	−	+	?	n.a.	−
AS	[Rac08]	−	+	+	−	−	+	+	+	−	−	+	+	+	+	+	+	−	?	?	n.a.	−
dMTS	[LV13a]	+	+	−	∼	−	?	n.a.	n.a.	−	−	+	+	+	+	+	?	n.a.	−	?	n.a.	−
DMTS	[LX90]	+	+	+	−	−	?	n.a.	n.a.	−	−	+	+	+	?	n.a.	?	n.a.	−	?	n.a.	−
DMTS	[BČK11]	+	+	+	−	−	+	+	+	−	−	+	+	∼	?	n.a.	?	n.a.	−	?	n.a.	−
DMTS	[Ben+13]	+	+	+	−	−	+	+	+	−	−	+	+	∼	+	+	+	n.a.	−	?	n.a.	−
1MTS	[FS08]	+	+	+	−	−	?	n.a.	n.a.	−	−	+	?	n.a.	?	n.a.	?	n.a.	−	?	n.a.	−
PMTS	[Ben+15]	+	+	+	−	+	?	n.a.	n.a.	−	−	+	?	n.a.	?	n.a.	?	n.a.	−	?	n.a.	−
PMTS'	[Ben+15]	−	+	+	−	+	?	n.a.	n.a.	−	−	+	?	n.a.	?	n.a.	?	n.a.	+	?	n.a.	−
mLTS	[LV11]	+	−	−	+	−	?	n.a.	n.a.	−	−	+	?	n.a.	?	n.a.	?	n.a.	?	?	n.a.	−
IA	[AH01a]	∼	∼	−	+	−	+	−	+	+	+	∼	−	n.a.	?	n.a.	−	n.a.	−	?	−	∼
IA	[LV13a]	∼	−	−	+	−	+	+	?	+	+	∼	+	+	+	+	−	n.a.	−	?	−	∼
IA	[BV14]	+	−	−	+	−	+	+	+	+	+	−	?	n.a.	?	n.a.	?	n.a.	−	?	−	∼
IA	[BR08]	−	−	−	−	−	+	+	+	+	+	∼	+	?	?	n.a.	+	+	−	?	−	∼
CS	[Chi13]	+	−	−	+	−	+	+	+	+	+	∼	+	+	+	+	∼	+	−	+	−	∼
IOMTS	[LNW07]	+	−	−	∼	−	+	−	+	+	+	−	−	n.a.	?	n.a.	−	n.a.	−	?	−	∼
MI	[Rac+11]	−	−	−	−	−	+	+	−	+	+	∼	+	+	?	n.a.	∼	−	+	+	−	∼
MIO	[Bau+10]	+	−	−	+	−	+	+	+	+	−	−	−	n.a.	?	n.a.	?	n.a.	−	?	−	∼
MIO	[HK15]	+	−	−	+	−	+	+	+	+	−	−	−	n.a.	?	n.a.	?	n.a.	−	?	−	∼
EMIO	[HK15]	+	−	−	+	−	?	n.a.	n.a.	n.a.	n.a.	n.a.	−	n.a.	?	n.a.	?	n.a.	−	?	+	+
MIA1	[LV13a]	∼	+	−	∼	−	+	+	+	+	+	−	+	+	+	+	?	n.a.	−	?	−	∼
MIA2o	[LVF15]	∼	+	−	+	−	+	+	+	+	+	−	+	+	+	+	?	n.a.	−	+	−	∼
MIA2p	[LVF15]	+	+	−	+	−	+	+	+	+	−	−	+	+	+	+	?	n.a.	−	+	−	∼
MIA3	[Buj+15]	+	+	−	+	−	+	+	+	+	+	∼	+	+	+	+	∼	+	−	+	−	∼
EMIA	[FL16b]	+	+	−	+	−	+	+	+	+	+	+	+	+	+	+	?	n.a.	−	?	+	+
EMIA	this thesis	+	+	−	+	−	+	+	+	+	+	+	+	+	+	+	∼	+	−	+	+	+

n.a. The feature is not applicable because it depends on a different feature that is either not supported (−) or not considered (?).

We discuss the table feature-wise:

N Nondeterminism arises ubiquitously in concurrent systems, e.g., due to races or abstraction. Some theories require input-determinism and, therefore, support nondeterminism only partially.

D Disjunctive transitions with single label ('d') or with multiple labels ('D'). Although not stated explicitly, the definition of refinement in [AH01a] makes that version of IA input deterministic with disjunctive input must-transitions. For the other theories, support of these features is obvious from the definitions.

τ Support of internal and weak transitions enables a theory to abstract from internal behaviour. This feature is obvious from the definitions.

P Global parameters permit one to express persistent choices. Only PMTS and its deterministic variant PMTS' support this feature.

Par A parallel composition operator expresses concurrency. Column '$\|$' shows whether a parallel operator is supported. Column '$\sqsubseteq$' denotes whether the refinement preorder is compositional with respect to $\|$. Compositionality is a strongly desired feature, and it is generally considered a bug if a theory is not compositional. Column 'a' shows associativity of $\|$. In general, the parallel operator of interface theories is a partial operation. In optimistic theories $\|$ is usually strongly associative, in pessimistic theories weakly.

Cp Compatibility of components may be considered optimistically, pessimistically or unanimously. Obviously, any optimistic theory may also consider pessimistic compatibility.

And A conjunction operator supports specifying components from different perspectives. Column 'glb' shows whether $\sqcap$ is the greatest lower bound with respect to the refinement preorder.

Or A disjunction operator completes a heterogeneous specification theory. Column 'lub' denotes whether $\sqcup$ is the least upper bound with respect to the refinement preorder.

Quo A quotient operator supports synthesis and reuse of component specifications. Column 'cp' denotes whether the quotient operator considers compatibility.

θ The refinement preorder is thorough. When based on modal refinement, this is usually the case for deterministic theories [Ben+15].

α Alphabet extension allows one to add new features to a specification and supports perspective-based specification when joined by conjunction.

Asm Support of multi-component assemblies enables one to check the compatibility of more than two components. This feature does not apply to the variants of MTS because MTS does not support compatibility.

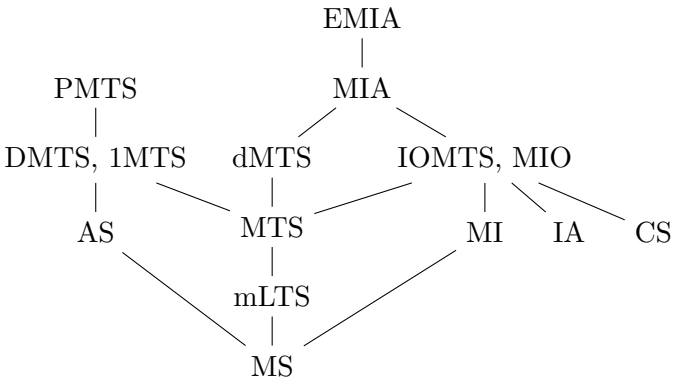


Figure 2.4.: Hasse diagram showing the expressiveness of specification theories, where expressiveness increases from bottom to top.

Err Error-awareness. Ignoring compatibility, the variants of MTS do not support this feature. The error-abstracting interface theories have partial implicit support of this feature. Full support is only present in error-preserving theories.

We may also compare specification theories with respect to their expressiveness as explained in Section 2.1.2. A Hasse diagram of this comparison is shown in Figure 2.4, where expressiveness increases from bottom to top. Most of the inclusions are obvious. In particular, any member of the MTS family may be included as the input-only fragment of an interface theory based on this member. We discuss the interesting cases of incomparable theories. DMTS and dMTS are incomparable because dMTS supports internal transitions and DMTS does not. Otherwise, DMTS would be more expressive than dMTS. MI and mLTS are incomparable because MI is deterministic and mLTS employs a coarser refinement preorder. MI and IA are incomparable because MI is deterministic and IA does not support modalities. We also included Chilton's *Component Specifications* (CS) [Chi13], which are similar to IA. However, IA and CS are incomparable because IA is more restrictive with respect to inconsistency and CS employs a coarser refinement preorder.

3. Modal Interface Automata

Modal Interface Automata [LV13a; LVF15; Buj+16; FL16b] belong to a family of interface theories that combine Interface Automata (IA) [AH01a] with Modal Transition Systems (MTS) [LT88]. They have been applied to conformance testing [LML15; LML17], to the parametrised refinement problem [SH15] and to the analysis of assume-guarantee rules [STH17].

In this thesis we distinguish two versions of Modal Interface Automata, namely *error-preserving Modal Interface Automata* (EMIA) and *error-abstracting Modal Interface Automata* (MIA).

From an axiomatic viewpoint, the difference between MIA and EMIA is as follows. The basic idea of errors e present in both variants of interface theories may be captured by the law $e \parallel p \equiv e$, expressing that a composed system is in an erroneous state if a component is. However, erroneous systems do not exist in the MIA semantics. Errors are abstracted away by universal states and, therefore, may be considered as models of *unknown* behaviour for which no guarantees can be made. Hence, they satisfy the law $e \sqsubseteq q \implies q \equiv e$, capturing the idea that an error cannot be introduced when refining an ordinary state. The only guarantee that this axiom gives is that a correct specification can only be refined to a correct implementation. However, a specified error may be refined (hence, redefined) to not being an error. In contrast, an error in EMIA models *unwanted* behaviour for which we know that it must not be implemented. This is captured by the additional law $q \sqsubseteq e \implies q \equiv e$, expressing that refinement cannot redefine an erroneous situation to be non-erroneous.

Historically, the theory MIA has been developed first [LV13a; LVF15; Buj+16; FL16b]. It is the first complete interface theory that combines IA and MTS without compromising standard properties such as compositionality of the refinement preorder or associativity of parallel composition in a nondeterministic setting with internal computations. In the tradition of IA, MIA includes the pruning of illegal states directly in the parallel composition operator. The theory EMIA has been developed afterwards in order to resolve several issues related to this error-abstraction of IA-based interface theories, which are unintuitive in practice (Section 3.3). In EMIA, errors have a semantic justification by themselves, and the error abstraction is an operator separate from parallel composition. However, because EMIA is one of the main contributions of this thesis and MIA may be seen as an abstraction of EMIA via a Galois insertion, it is more compelling to present EMIA first. Large parts of the work presented in this chapter are published in [Buj+15; FL16a; FL16b].

The author's own contributions presented in this chapter are as follows: The error-preserving interface theory EMIA that solves all of Gaps 1–7 listed in Section 1.1 yielding the most general interface theory to date; a general definition of the concept of alphabet

extension subsuming the concrete alphabet extension operators that are considered in the literature; an investigation of different alphabet extension operators for interface theories shows that IA-based refinement is insufficient for perspective-based specification; a quotient operator for MIA and EMIA, which is the first one supporting nondeterministic dividends and compatibility; a detailed discussion of quotienting including several properties of the quotient, the difficulties of generalising the quotient to nondeterministic divisors, and the proof that relaxing the alphabet conditions on the operands renders the quotient nonunique for all interface theories based on IA; an investigation of error-preserving and error-abstracting interface theories relating them by a Galois insertion that results in a better understanding of error-abstraction in interface theories; an investigation of implication and negation for interface theories and a proof that implication and negation cannot be provided in practice.

3.1. Error-preserving Modal Interface Automata

Our interface theory *Error-preserving Modal Interface Automata* (EMIA), which we present in this section, is equipped with a *parallel composition operator* modelling concurrency and communication, a *conjunction operator* permitting the specification of a component from different perspectives, a *disjunction operator* for providing alternatives, a *quotienting operator* allowing for component reuse, and a *compositional refinement preorder* enabling the substitution of an interface by a more concrete version. In addition to these standard requirements on interface theories, EMIA solves Issues I1–I4 presented in Section 2.3.3 and discussed in Section 3.3. We achieve this by introducing *fatal error states*, which represent unresolvable incompatibilities between interfaces, in contrast to illegal states in IA, which may be resolved in a refinement. This enables EMIA to deal with errors on a semantic level, because forbidden behaviour can be modelled by input transitions leading to a fatal error state.

3.1.1. Basic Definitions

Definition 3.1 (Error-preserving Modal Interface Automata)**.** An *Error-preserving Modal Interface Automaton* (EMIA) is a tuple $P := (S_P, I_P, O_P, \longrightarrow_P, \dashrightarrow_P, S_P^0, E_P, U_P)$, where S_P is the set of states, I_P, O_P are the disjoint alphabets of input and output actions not including the silent action τ (we define $A_P := I_P \cup O_P$ and $\Omega_P := O_P \cup \{\tau\}$), $\longrightarrow_P \subseteq S_P \times (A_P \cup \{\tau\}) \times \mathfrak{P}(S_P)$ is the disjunctive must-transition relation, $\dashrightarrow_P \subseteq S_P \times (A_P \cup \{\tau\}) \times S_P$ is the may-transition relation, $S_P^0 \subseteq S_P$ is the set of initial states, $E_P \subseteq S_P$ is the set of fatal error states and $U_P \subseteq S_P$ is the set of *universal states*, if the following conditions hold:

E1. For all $\alpha \in A_P \cup \{\tau\}$ and $p \xrightarrow{\alpha} P'$, we have $\forall p' \in P' .\, p \overset{\alpha}{\dashrightarrow} p'$, (syntactic consistency)
E2. States in $E_P \cup U_P$ have no outgoing transitions, (sink condition)
E3. $E_P \cap U_P = \emptyset$. (exclusive markings)

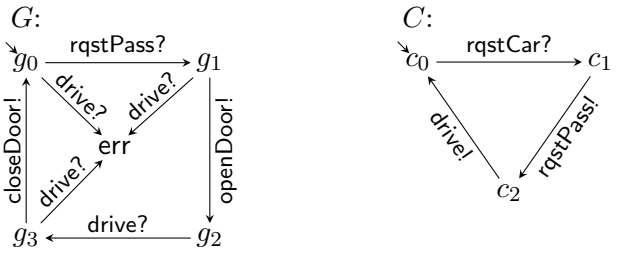


Figure 3.1.: Running example of a driving assistance system including a garage G and a car C with alphabets $A_G := \{\mathsf{rqstPass?}, \mathsf{drive?}\}/\{\mathsf{closeDoor!}\}$ and $A_C := \{\mathsf{rqstCar?}\}/\{\mathsf{rqstPass!}, \mathsf{drive!}\}$, respectively.

If $S_P^0 = \emptyset$, then P is called *inconsistent* and often denoted as $\bot$. An EMIA P is an *implementation* (i.e., an LTS) if $|S_P^0| = 1$, $|P'| = 1$ for all $p \xrightarrow{\alpha} P'$ and, for each $p \dashrightarrow^{\alpha} p'$, there is a $p \xrightarrow{\alpha} \{p'\}$. We define $\mathrm{must}(p, \alpha) := \bigcup\{P' \in S_P \mid p \xrightarrow{\alpha} P'\}$ and $\mathrm{may}(p, \alpha) := \{p' \in S_P \mid p \dashrightarrow^{\alpha} p'\}$.

In the following we often omit the index P when referring to components of an EMIA P, e.g., we write I for I_P. Similarly, we write, e.g., I_1 instead of I_{P_1} for EMIA P_1. In addition, we let i, o, a, ω and α stand for representatives of the alphabets I, O, A, Ω and $A \cup \{\tau\}$, respectively; we write $A = I/O$ when highlighting inputs I and outputs O in an alphabet A. In the context of weak transitions that abstract from τs, we use the notation $\hat{\alpha}$, where $\hat{\alpha} := a$ if $\alpha = a \neq \tau$ and $\hat{\alpha} := \epsilon$ if $\alpha = \tau$. In figures, we often refer to an action a as $a?$ if $a \in I$, and $a!$ if $a \in O$. Must-transitions (may-transitions) are drawn using solid, possibly splitting arrows (dashed arrows); any depicted must-transition also implicitly represents the underlying may-transition(s) due to syntactic consistency. For notational convenience, we let $p \xrightarrow{a} p'$, $p \not\xrightarrow{a}$ and $p \dashrightarrow\!\!\!\!\!\!/\;^{a}$ denote $p \xrightarrow{a} \{p'\}$, $\nexists P'.\ p \xrightarrow{a} P'$ and $\nexists p'.\ p \dashrightarrow^{a} p'$, respectively.

In contrast to [FL16b] we include universal states in the definition of EMIA for several reasons. First, because the quotient is the maximal specification Q satisfying $Q \otimes D \sqsubseteq_{\mathrm{e}} P$, an action that is not used by D may be implemented with arbitrary subsequent behaviour in Q, which is exactly the meaning of input transitions leading to universal states. Second, as universal states are already necessary in MIA in order to make the refinement preorder compositional when pruning errors [Buj+16], allowing universal states in EMIA yields a more uniform presentation of the two theories. As a side benefit, the Galois insertion of MIA into EMIA becomes simpler because an infinite disjunction is not needed anymore (see Section 3.2). Note that one may choose to replace the sets U and E by single elements $\top$ and err without changing the underlying semantics of the theory. We decided to employ sets instead of single elements because this representation is easier to be implemented in a software tool.

Example 3.2. As a running example, we consider a driving assistance system that enables a car to drive into and out of a garage autonomously. Such a system must communicate with the garage in order to make it open and close its door. Figure 3.1

Figure 3.2.: Examples of weak transitions and weak refinement.

shows a specification G of the garage's interface. Starting in state g_0, the garage is ready to receive a passage request (rqstPass?). After such a request, the garage opens its door (openDoor!), waits for a car driving in or out (drive?) and, finally, closes the door (closeDoor!) again. The fatal error state err reachable via drive?-transitions from states g_0, g_1 and g_3 expresses the safety requirement that driving into or out of the garage is undesired when the garage door is closed or about to be closed. The car C starts in state c_0 waiting for a user's request (rqstCar?). Upon receiving such a request, the car requests passage from the garage (rqstPass!) and then drives into or out of the garage (drive!), reaching state c_0 again.

Next, we define weak transitions that abstract from internal behaviour. Our definition of weak transitions is adopted from MIA [Buj+16]:

Definition 3.3 (Weak Transition Relations)**.** Let P be an EMIA. We define *weak* must- and may-transition relations, $\rightarrowtail\!\!\!\!\!\rightarrow$ and $\succ\!\!\!\!\rightarrow$ respectively, as the smallest relations satisfying the following conditions, where we use $P' \overset{\hat{\alpha}}{\rightarrowtail\!\!\!\!\!\rightarrow} P''$ as a shorthand for $\forall p{\in}P'.\, \exists P_p.\, p \overset{\hat{\alpha}}{\rightarrowtail\!\!\!\!\!\rightarrow} P_p \wedge P'' = \bigcup_{p\in P'} P_p$:

WT1. $p \overset{\epsilon}{\rightarrowtail\!\!\!\!\!\rightarrow} \{p\}$ for all $p \in S_P$,
WT2. $p \xrightarrow{\tau} P'$ and $P' \overset{\hat{\alpha}}{\rightarrowtail\!\!\!\!\!\rightarrow} P''$ implies $p \overset{\hat{\alpha}}{\rightarrowtail\!\!\!\!\!\rightarrow} P''$,
WT3. $p \xrightarrow{a} P'$ and $P' \overset{\epsilon}{\rightarrowtail\!\!\!\!\!\rightarrow} P''$ implies $p \overset{a}{\rightarrowtail\!\!\!\!\!\rightarrow} P''$,
WT4. $p \overset{\epsilon}{\succ\!\!\!\!\rightarrow} p$,
WT5. $p \overset{\epsilon}{\succ\!\!\!\!\rightarrow} p'' \overset{\tau}{\dashrightarrow} p'$ implies $p \overset{\epsilon}{\succ\!\!\!\!\rightarrow} p'$,
WT6. $p \overset{\epsilon}{\succ\!\!\!\!\rightarrow} p'' \overset{\alpha}{\dashrightarrow} p''' \overset{\epsilon}{\succ\!\!\!\!\rightarrow} p'$ implies $p \overset{\alpha}{\succ\!\!\!\!\rightarrow} p'$.

We write $\overset{a}{\longrightarrow\!\!\!\!\!\rightarrow}$ for transitions that are built up according to WT3 and call them *trailing-weak* must-transitions. Similarly, $\overset{a}{\dashrightarrow}$ stands for trailing-weak may-transitions.

Example 3.4. For examples of weak transitions, consider the EMIA on the left-hand side of Figure 3.2. By applying WT1 and WT2 of Definition 3.3, any τ-transition is also a weak ϵ-transition. Similarly, every a-transition is also a weak a-transition by WT1 and WT3. Transition $2 \xrightarrow{\tau} \{4,5\}$ can be extended to $2 \overset{o}{\rightarrowtail\!\!\!\!\!\rightarrow} \{7,8\}$ by applying WT2. Hence, $0 \xrightarrow{\tau} \{1,2\}$ extends to $0 \overset{o}{\rightarrowtail\!\!\!\!\!\rightarrow} \{3,7,8\}$. Observe that our weak must-transitions correspond to standard weak transitions of LTS in the case that only must-transitions with a single target state are used.

We employ dMTS instead of DMTS (see Section 2.2.3), i.e., we require that all branches of a disjunctive transition have the same label. We do so because dMTS is sufficient for our purposes and does away with technical complications of parallel composition in the presence of weak transitions. In dMTS, a disjunctive must-transition guarantees the existence of an action, which is exploited in the definition of weak transitions, e.g., in WT2. Such a guarantee is not provided in DMTS. In a DMTS-rule similar to WT2, α would represent a set of possible labels and could be implemented differently for each $p' \in P'$. Hence, the existence of weak transitions of a specification depends on the transitions of its implementations. In particular, τ may be included in the set of possible labels.

This issue becomes even more complicated in the context of parallel composition. The usual way of defining parallel composition on DMTS, e.g., as is done in [BČK10], is by unfolding each disjunctive must-transition into its set of possible implementation variants, i.e., selections of transition branches. The parallel composition of two components is then obtained by forming all pairwise products of the components' implementation variants. The unfolding operation corresponds to a transformation of a conjunctive normal form into a disjunctive normal form and, thus, is only a change of representation. However, in order to define weak transitions in the unfolded representation, one has to unfold the τ-closure of each transition. If τ-loops are involved, this may result in an infinite unfolding—even in case of finite DMTS—because a different implementation may be chosen in each iteration of the loop.

3.1.2. Refinement

Our *error-preserving modal refinement preorder* $\sqsubseteq_{\mathrm{e}}$ is adapted from standard modal refinement of MTS (see Section 2.2.3 and [Lar89; LX90]). Intuitively, $P \sqsubseteq_{\mathrm{e}} Q$ for EMIAs P and Q enforces that P's may-transitions are permitted by Q while for any of Q's disjunctive must-transitions at least one of the branches is implemented by P.

In addition, error-preserving modal refinement reflects the meaning of universal states that may be refined by everything, as well as the meaning of fatal error states that may only be refined by states that are erroneous themselves. In other words, universal states are reflected and fatal error states are reflected *and* preserved by $\sqsubseteq_{\mathrm{e}}$.

Note that our definition of refinement (as well as conjunction and disjunction) is for interfaces with equal alphabets; this is not a restriction because we consider alphabet extension later.

Definition 3.5 (Error-preserving Modal Refinement)**.** Let P and Q be EMIAs with equal alphabets, i.e., $I_P = I_Q$ and $O_P = O_Q$. A relation $\mathcal{R} \subseteq S_P \times S_Q$ is an *error-preserving modal refinement relation* if, for all $(p, q) \in \mathcal{R}$ with $q \notin U_Q$,

R1. $p \in E_P$ iff $q \in E_Q$,
R2. $p \notin U_P$,
R3. $q \xrightarrow{i}_Q Q'$ implies $\exists P'.\, p \xrightarrow{i}\!\!\!\!\rightarrow_P P' \wedge \forall p' \in P'\ \exists q' \in Q'.\, (p', q') \in \mathcal{R}$,
R4. $q \xrightarrow{\omega}_Q Q'$ implies $\exists P'.\, p \rightarrowtail\!\!\!\!\rightarrow_P P' \wedge \forall p' \in P'\ \exists q' \in Q'.\, (p', q') \in \mathcal{R}$,

R5. $p \dashrightarrow^{i}_{P} p'$ implies $\exists q'.\, q \dashrightarrow\!\!\!\!\!\!\rightarrow^{i}_{Q} q' \wedge (p', q') \in \mathcal{R}$,
R6. $p \dashrightarrow^{\omega}_{P} p'$ implies $\exists q'.\, q \succ\!\dashrightarrow\!\!\!\!\!\!\rightarrow^{\omega}_{Q} q' \wedge (p', q') \in \mathcal{R}$.

We write $p \sqsubseteq_{\mathrm{e}} q$ if there is a refinement relation $\mathcal{R}$ with $(p, q) \in \mathcal{R}$. Similarly, we write $P \sqsubseteq_{\mathrm{e}} Q$ if, for all $p \in S^0_P$, there is a $q \in S^0_Q$ with $p \sqsubseteq_{\mathrm{e}} q$. If $p \sqsubseteq_{\mathrm{e}} q$ and $q \sqsubseteq_{\mathrm{e}} p$, we employ the symbol $p \equiv_{\mathrm{e}} q$, and similar for EMIAs P, Q.

Note that $\mathsf{err} \sqsubseteq_{\mathrm{e}} \top$ for all $\mathsf{err} \in E$ and $\top \in U$, i.e., universal states also include the possibility of error.

An example of a refinement can be found in Figure 3.2, where the left EMIA refines the right one due to the refinement relation $\{(0, 0'), (1, 0'), (2, 0'), (4, 0'), (5, 0'), (3, 1'), (7, 1'), (8, 1'), (6, 2')\}$. Observe how the refined states 3 and 7 (and 8) of state $1'$ implement the outgoing $i?$-may-transition differently.

Introducing leading τs during refinement is not permitted for an input must-transition $p \xrightarrow{i}\!\!\!\!\rightarrow_P P'$ in Rule R3 because, otherwise, the guarantee of being ready for input i that is expressed by transition $q \xrightarrow{i}_Q Q'$ would not be preserved and a new communication mismatch would be introduced. In a pure EMIA setting we may relax Rule R5 by permitting leading τ-transitions in Q. In MIA, this relaxation would break compositionality with respect to parallel composition due to the employed pruning operation. Therefore, we do without this relaxation in order to make the two theories more comparable. In particular, we employ the same pruning operation for establishing the Galois insertion between MIA and EMIA.

The refinement relation $\sqsubseteq_{\mathrm{e}}$ is a preorder:

Lemma 3.6 ($\sqsubseteq_{\mathrm{e}}$ is a Preorder)**.** *Error-preserving modal refinement $\sqsubseteq_{\mathrm{e}}$ is reflexive and transitive.*

Proof sketch. Reflexivity is easy because the identity relation is an isomorphism, which trivially satisfies all refinement conditions. The proof of transitivity closely follows the proof of Bujtor and Vogler in [Buj+16]; therefore, we only sketch the proof idea: given EMIAs P, Q, R with refinement relations $\mathcal{R}_{PQ}$ and $\mathcal{R}_{QR}$, we have to show that $\mathcal{R} := \{(p, r) \mid \exists q \in S_Q.\, (p, q) \in \mathcal{R}_{PQ} \wedge (q, r) \in \mathcal{R}_{QR}\}$ is an error-preserving modal refinement relation. It is easy to see that conditions R1 and R2 hold transitively. If a relation satisfies conditions R3 through R6, then it also satisfies the same conditions with weak transitions in the premises, e.g., a rule R3' of the form $q \xrightarrow{i}\!\!\!\!\rightarrow Q'$ implies $\exists P'.\, p \xrightarrow{i}\!\!\!\!\rightarrow P'$ and $\forall p' \in P'\ \exists q' \in Q'.\, (p', q') \in \mathcal{R}$. Hence, these conditions also follow transitively. □

3.1.3. Parallel Composition

IA's parallel composition operator synchronises input and output transitions to τ-transitions. In contrast, we define a multicast parallel composition, where an output can synchronise with multiple input transitions as in MI [Rac+11] and MIA [Buj+16].

Definition 3.7 (Parallel Composition)**.** Let P and Q be EMIAs. We call P and Q *composable* if $O_P \cap O_Q = \emptyset$. If P and Q are composable, their *error-preserving multicast*

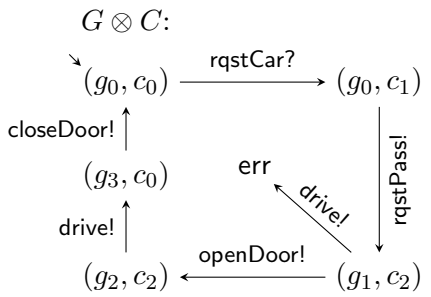


Figure 3.3.: Parallel composition $G \otimes C$ of the specifications in Figure 3.1, with $A := \{\mathsf{rqstCar?}\}/\{\mathsf{rqstPass!}, \mathsf{drive!}, \mathsf{openDoor!}, \mathsf{closeDoor!}\}$.

parallel composition $P \otimes Q$ is defined by $S_{P\otimes Q} := S_P \times S_Q$, $I_{P\otimes Q} := (I_P \cup I_Q) \setminus O_{P\otimes Q}$, $O_{P\otimes Q} := O_P \cup O_Q$, $S^0_{P\otimes Q} := S^0_P \times S^0_Q$, $E_{P\otimes Q} := (E_P \times S_Q) \cup (S_P \times E_Q)$, $U_{P\otimes Q} := ((S_P \setminus E_P) \times U_Q) \cup (U_P \times (S_Q \setminus E_Q))$, and the transition relations are given by the following rules:

P1. $(p,q) \xrightarrow{\alpha} P' \times \{q\}$ if $p \xrightarrow{\alpha} P'$ and $\alpha \notin A_Q$,
P2. $(p,q) \xrightarrow{\alpha} \{p\} \times Q'$ if $\alpha \notin A_P$ and $q \xrightarrow{\alpha} Q'$,
P3. $(p,q) \xrightarrow{a} P' \times Q'$ if $p \xrightarrow{a} P'$ and $q \xrightarrow{a} Q'$ for some $a \in A_P \cap A_Q$,
P4. $(p,q) \dashrightarrow^{\alpha} (p',q)$ if $p \dashrightarrow^{\alpha} p'$ and $\alpha \notin A_Q$,
P5. $(p,q) \dashrightarrow^{\alpha} (p,q')$ if $\alpha \notin A_P$ and $q \dashrightarrow^{\alpha} q'$,
P6. $(p,q) \dashrightarrow^{a} (p',q')$ if $p \dashrightarrow^{a} p'$ and $q \dashrightarrow^{a} q'$ for some $a \in A_P \cap A_Q$.

We also write $p \otimes q$ for (p,q).

Note that an error in one component implies an error in the overall system, whereas universal behaviour in one component extends to the overall system only in absence of errors.

IA-based interface theories usually define a communication mismatch for p at q as a situation where an action $a \in O_P \cap I_Q$ is permitted at p and not required at q. In EMIA, such an optional input transition, which may be refined to required or forbidden behaviour, is expressed as a disjunctive must-transition containing a fatal error state in its set of target states. For example, optional $a?$-transitions from q to states q_1 and q_2 are modelled as $q \xrightarrow{a?} \{q_1, q_2, q_3\}$ for some fatal error state $q_3 \in E_Q$.

Example 3.8. Figure 3.3 shows the parallel composition $G \otimes C$ of the garage's and the car's specifications from Figure 3.1. The components synchronise on their shared actions (rqstPass, drive, openDoor, closeDoor) by P3, while the unshared action rqstCar is interleaved due to P2.

It is easy to see that parallel composition is associative and commutative. Further, $\sqsubseteq_e$ is a precongruence with respect to $\otimes$:

Theorem 3.9 (Compositionality)**.** *If P_1, P_2 and Q are EMIAs such that $P_1 \sqsubseteq_e P_2$ and P_2, Q are composable, then P_1 and Q are composable and $P_1 \otimes Q \sqsubseteq_e P_2 \otimes Q$.*

Proof. We write I_P, O_P and A_P for the equal alphabets of P_1 and P_2. Composability is trivial. We show that $\mathcal{R} := \{(p_1 \otimes q, p_2 \otimes q) \mid p_1 \sqsubseteq_e p_2\}$ is an error-preserving modal refinement relation. For $(p_1 \otimes q, p_2 \otimes q) \in \mathcal{R}$ with $p_2 \otimes q \notin U_{P \otimes Q}$, we consider the following cases:

R1 $p_1 \otimes q \notin E_{P_1 \otimes Q}$ iff (by Definition 3.7) $p_1 \notin E_{P_1} \wedge q \notin E_Q$ iff (by $p_1 \sqsubseteq_e p_2$ and R1) $p_2 \notin E_{P_2} \wedge q \notin E_Q$ iff (by Definition 3.7) $p_2 \otimes q \notin E_{P_2 \otimes Q}$.

R2 We consider two cases:

1. $(p_2, q) \in E_{P_2 \otimes Q}$: As shown for Case R1, we have $(p_1, q) \in E_{P_1 \otimes Q}$ and, by Definition 3.7 and E3, $(p_1, q) \notin U_{P_1 \otimes Q}$.
2. $(p_2, q) \notin E_{P_2 \otimes Q}$: By $(p_2, q) \notin U_{P_2 \otimes Q}$, we have $p_2 \notin U_{P_2}$ and $q \notin U_Q$. Then, $p_1 \sqsubseteq_e p_2$ implies $p_1 \notin U_{P_1}$; hence, $(p_1, q) \notin U_{P_1 \otimes Q}$.

R3 Let $p_2 \otimes q \xrightarrow{i} R$ due to one of P1, P2 or P3:

- **P1** $R = P_2' \times \{q\}$ for some transition $p_2 \xrightarrow{i} P_2'$. By $p_1 \sqsubseteq_e p_2$, there is a $p_1 \xrightarrow{i} \!\!\!\!\rightarrow P_1'$ such that, for all $p_1' \in P_1'$, there exists a $p_2' \in P_2'$ with $p_1' \sqsubseteq_e p_2'$. Thus, we have $(p_1' \otimes q, p_2' \otimes q) \in \mathcal{R}$, and P1 implies $p_1 \otimes q \xrightarrow{i} \!\!\!\!\rightarrow P_1' \times \{q\}$.
- **P2** $R = \{p_2\} \times Q'$ for some $q \xrightarrow{i} Q'$. By P2 we have $p_1 \otimes q \xrightarrow{i} \{p_1\} \times Q'$, and $p_1 \sqsubseteq_e p_2$ implies $(p_1 \otimes q', p_2 \otimes q') \in \mathcal{R}$ for all $q' \in Q'$.
- **P3** $R = P_2' \times Q'$ due to $p_2 \xrightarrow{i} P_2'$ and $q \xrightarrow{i} Q'$. The argument is analogous to that of case P1, when replacing the application of P1 by P3 in the last step.

R4 Analogous to R3.

R5 Let $p_1 \otimes q \dashrightarrow^{i} p_1' \otimes q'$ due to one of the rules P4, P5 or P6:

- **P4** $q' = q$ for a transition $p_1 \dashrightarrow^{i} p_1'$. By $p_1 \sqsubseteq_e p_2$, there is a $p_2 \dashrightarrow^{i\,*} p_2'$ such that $p_1' \sqsubseteq_e p_2'$. Thus, we have $(p_1' \otimes q, p_2' \otimes q) \in \mathcal{R}$, and P4 implies $p_2 \otimes q \dashrightarrow^{i\,*} p_2' \otimes q$.
- **P5** $p_1' = p_1$ for some $q \dashrightarrow^{i} q'$. By P5, we have $p_2 \otimes q \dashrightarrow^{i} p_2 \otimes q'$, and $p_1 \sqsubseteq_e p_2$ implies $(p_1 \otimes q', p_2 \otimes q') \in \mathcal{R}$.
- **P6** $R = P_1' \times Q'$ due to $p_1 \dashrightarrow^{i} P_1'$ and $q \dashrightarrow^{i} Q'$. The argument is similar to that of case P4, where the application of P4 is replaced by P6 in the last step.

R6 Analogous to R5. □

3.1.4. Hiding and Restriction

We now define the standard process algebraic operators *hiding* [Hoa85] and *restriction* [Mil89], which are easily adapted from [Buj+15]. When scoping output actions, these are still performed because they are under the control of the system. However, the action is no longer observable from outside of the scope. Hence, the action is internal,

i.e., a hiding operator is employed. In contrast, input actions are only performed because of an outside stimulus. Scoping an input action blocks the system from listening to this stimulus. Therefore, scoped input actions are removed, i.e., a restriction operator is employed. A similar idea is used in the IA-setting of [CJK13], where hiding and restriction are combined into a single operation.

Definition 3.10 (Hiding)**.** Let $P = (S_P, I_P, O_P, \longrightarrow_P, \dashrightarrow_P, S_P^0, E_P, U_P)$ be an EMIA and L a set of actions with $L \cap I_P = \emptyset$. We define P *hiding* L as the EMIA $P \,/\, L := (S_P, I_P, O \setminus L, \longrightarrow_{P/L}, \dashrightarrow_{P/L}, S_P^0, E_P, U_P)$, where all transition labels $o \in L$ are replaced by τ.

Definition 3.11 (Restriction)**.** Let $P = (S_P, I_P, O_P, \longrightarrow_P, \dashrightarrow_P, S_P^0, E_P, U_P)$ be an EMIA and L a set of actions with $L \cap O_P = \emptyset$. We define P *restricted* L as the EMIA $P \setminus L := (S_P, I_P \setminus L, O_P, \longrightarrow_{P\setminus L}, \dashrightarrow_{P\setminus L}, S_P^0, E_P, U_P)$, where all transitions with a label contained in L are removed.

Observe that hiding and restriction yield well-defined EMIAs.

Lemma 3.12 (Weak Must-Transitions under Hiding)**.** *Let P be an EMIA, $L \cap I_P = \emptyset$ and $o \in L \cap O_P$. If $p \overset{o}{\rightarrowtail\!\!\!\!\!\twoheadrightarrow}_P P'$, then $p \overset{\epsilon}{\rightarrowtail\!\!\!\!\!\twoheadrightarrow}_{P/L} P'$.*

Proof. By induction on the definition of $p \overset{o}{\rightarrowtail\!\!\!\!\!\twoheadrightarrow}_P P'$. If $p \overset{o}{\rightarrowtail\!\!\!\!\!\twoheadrightarrow}_P P'$ is due to WT3 of Definition 3.3, then the claim is obvious. Otherwise, $p \overset{o}{\rightarrowtail\!\!\!\!\!\twoheadrightarrow}_P P'$ is due to some $p \xrightarrow{\tau}_P \bar{P}$ and $\bar{P} \overset{o}{\rightarrowtail\!\!\!\!\!\twoheadrightarrow}_P P'$ according to WT2. By induction hypothesis, we have $\bar{p} \overset{\epsilon}{\rightarrowtail\!\!\!\!\!\twoheadrightarrow}_{P/L} P_{\bar{p}}$ for each $\bar{p} \in \bar{P}$ and $P' = \bigcup_{\bar{p}\in\bar{P}} P_{\bar{p}}$. By WT2, we obtain $p \overset{\epsilon}{\rightarrowtail\!\!\!\!\!\twoheadrightarrow}_{P/L} P'$. □

As desired, EMIA-refinement is a precongruence with respect to hiding and restriction:

Proposition 3.13. *Let P, Q be EMIAs with equal alphabets and $P \sqsubseteq_{\mathrm{e}} Q$.*

1. *$P \,/\, L \sqsubseteq_{\mathrm{e}} Q \,/\, L$ for any set L of actions excluding τ with $L \cap I = \emptyset$.*
2. *$P \setminus L \sqsubseteq_{\mathrm{e}} Q \setminus L$ for any set L of actions excluding τ with $L \cap O = \emptyset$.*

Proof. Since $P \sqsubseteq_{\mathrm{e}} Q$, there is an EMIA-refinement relation $\mathcal{R}$ with $(p, q) \in \mathcal{R}$. We show that $\mathcal{R}$ is also an EMIA-refinement relation for $P \,/\, L \sqsubseteq_{\mathrm{e}} Q \,/\, L$ and $P \setminus L \sqsubseteq_{\mathrm{e}} Q \setminus L$. The only interesting case concerns hiding and Rule R4 of Definition 3.5, i.e., $q \xrightarrow{\tau}_{Q/L} Q'$ due to $q \xrightarrow{o}_Q Q'$ for $o \in O \cap L$. The latter is matched by a transition $p \overset{o}{\rightarrowtail\!\!\!\!\!\twoheadrightarrow}_P P'$ with $\forall p' \in P' \,\exists q' \in Q'.\, (p', q') \in \mathcal{R}$. By Lemma 3.12, this yields $p \overset{\epsilon}{\rightarrowtail\!\!\!\!\!\twoheadrightarrow}_{P/L} P'$. □

Originally, IA employs a parallel composition with immediate hiding [AH01a]. This can easily be expressed by combining our parallel composition and the hiding operator, such that $P \mid Q = (P \otimes Q) \,/\, S$, where S is the set of synchronising actions. However, the immediate hiding weakens the associativity of this composition operation, e.g., if $a \in I_P \cap O_Q \cap O_R$, then $(P \mid Q) \mid R$ may be composed while $P \mid (Q \mid R)$ is uncomposable. We omit the details here, because they are presented in [Buj+16] for MIA and may directly be adopted to EMIA.

3.1.5. Conjunction

Perspective-based specification is concerned with specifying a system component from separate perspectives, such that the component satisfies each of these perspective specifications. For example, each requirement for a component may describe a perspective. The component's overall specification is the most general specification refining all perspective specifications, i.e., it is the greatest lower bound with respect to the refinement preorder. This conjunction operator is defined in two stages. In the first stage, the *conjunctive product* defines the combined requirements of the conjuncts. In the second stage, *inconsistent states* that capture contradictory requirements are removed.

Definition 3.14 (Conjunctive Product)**.** Let P, Q be EMIAs with equal alphabets. The *conjunctive product* of P and Q is $P \,\&\, Q := (S_{P\&Q}, I, O, \longrightarrow_{P\&Q}, \dashrightarrow_{P\&Q}, S^0_{P\&Q}, E_{P\&Q}, U_{P\&Q})$ with $S_{P\&Q} := S_P \times S_Q$, $S^0_{P\&Q} := S^0_P \times S^0_Q$, $E_{P\&Q} := (E_P \times (E_Q \cup U_Q)) \cup ((E_P \cup U_P) \times E_Q)$, $U_{P\&Q} := U_P \times U_Q$, and the transition relations are given by the following rules:

C1.	$(p,q) \xrightarrow{i} \{(p',q') \mid p' \in P',\ q \overset{i}{\dashrightarrow\!\!\!\!\!\!\rightarrow} q'\}$	if $p \xrightarrow{i} P'$ and $q \overset{i}{\dashrightarrow\!\!\!\!\!\!\rightarrow}$,
C2.	$(p,q) \xrightarrow{i} \{(p',q') \mid p \overset{i}{\dashrightarrow\!\!\!\!\!\!\rightarrow} p',\ q' \in Q'\}$	if $p \overset{i}{\dashrightarrow\!\!\!\!\!\!\rightarrow}$ and $q \xrightarrow{i} Q'$,
C3.	$(p,q) \xrightarrow{\omega} \{(p',q') \mid p' \in P',\ q \overset{\omega}{\succ\!\dashrightarrow\!\!\!\!\!\!\rightarrow} q'\}$	if $p \xrightarrow{\omega} P'$ and $q \overset{\omega}{\succ\!\dashrightarrow\!\!\!\!\!\!\rightarrow}$,
C4.	$(p,q) \xrightarrow{\omega} \{(p',q') \mid p \overset{\omega}{\succ\!\dashrightarrow\!\!\!\!\!\!\rightarrow} p',\ q' \in Q'\}$	if $p \overset{\omega}{\succ\!\dashrightarrow\!\!\!\!\!\!\rightarrow}$ and $q \xrightarrow{\omega} Q'$,
C5.	$(p,q) \xrightarrow{\alpha} P' \times \{q\}$	if $p \xrightarrow{\alpha} P'$ and $q \in U_Q$,
C6.	$(p,q) \xrightarrow{\alpha} \{p\} \times Q'$	if $p \in U_P$ and $q \xrightarrow{\alpha} Q'$,
C7.	$(p,q) \overset{i}{\dashrightarrow} (p',q')$	if $p \overset{i}{\dashrightarrow\!\!\!\!\!\!\rightarrow} p'$ and $q \overset{i}{\dashrightarrow\!\!\!\!\!\!\rightarrow} q'$,
C8.	$(p,q) \overset{\omega}{\dashrightarrow} (p',q')$	if $p \overset{\omega}{\succ\!\dashrightarrow\!\!\!\!\!\!\rightarrow} p'$ and $q \overset{\omega}{\succ\!\dashrightarrow\!\!\!\!\!\!\rightarrow} q'$,
C9.	$(p,q) \overset{\tau}{\dashrightarrow} (p',q)$	if $p \overset{\tau}{\succ\!\dashrightarrow\!\!\!\!\!\!\rightarrow} p'$,
C10.	$(p,q) \overset{\tau}{\dashrightarrow} (p,q')$	if $q \overset{\tau}{\succ\!\dashrightarrow\!\!\!\!\!\!\rightarrow} q'$,
C11.	$(p,q) \overset{\alpha}{\dashrightarrow} (p',q)$	if $p \overset{\alpha}{\dashrightarrow} p'$ and $q \in U_Q$,
C12.	$(p,q) \overset{\alpha}{\dashrightarrow} (p,q')$	if $p \in U_P$ and $q \overset{\alpha}{\dashrightarrow} q'$,

A state (p,q) of $P \,\&\, Q$ is a candidate for refining both p and q. Because (p,q) cannot simultaneously require and forbid the same action a or be at once fatal and non-fatal, some states p and q do not have a common refinement. In such cases, (p,q) is called (logically) *inconsistent* and has to be removed from the candidates, which also includes the removal of all states that require transitions leading to inconsistent states. In order to be the greatest common refinement of p and q, a state (p,q) may only be erroneous if p and q are erroneous or universal. This explains the definition of $E_{P\&Q}$ which obviously must exclude $U_{P\&Q}$.

Definition 3.15 (Conjunction)**.** The set $F_\& \subseteq S_{P\&Q}$ of *logically inconsistent* states is defined as the smallest set satisfying the following rules:

CF1.	$(p,q) \in (E_P \times (S_Q \setminus (E_Q \cup U_Q))) \cup ((S_P \setminus (E_P \cup U_P)) \times E_Q)$	implies $(p,q) \in F_\&$,
CF2.	$(p,q) \notin E_{P\&Q} \cup U_{P\&Q}$, $p \xrightarrow{i}$ and $q \not\overset{i}{\dashrightarrow}$	implies $(p,q) \in F_\&$,

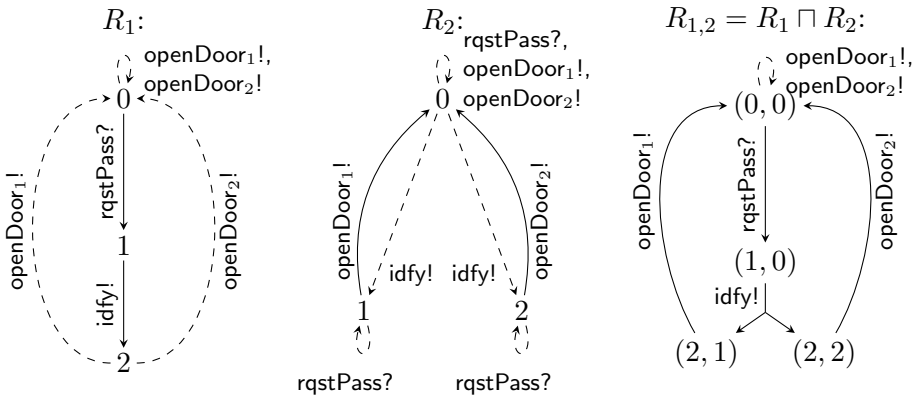


Figure 3.4.: Example of conjunction with the alphabet {rqstPass?}/{idfy!, openDoor$_1$!, openDoor$_2$!}.

CF3. $(p,q) \notin E_{P\&Q} \cup U_{P\&Q}$, $p \not\xrightarrow{i}$ and $q \xrightarrow{i}$ implies $(p,q) \in F_{\&}$,
CF4. $(p,q) \notin E_{P\&Q} \cup U_{P\&Q}$, $p \xrightarrow{\omega}$ and $q \not\overset{\omega}{\twoheadrightarrow}$ implies $(p,q) \in F_{\&}$,
CF5. $(p,q) \notin E_{P\&Q} \cup U_{P\&Q}$, $p \not\overset{\omega}{\twoheadrightarrow}$ and $q \xrightarrow{\omega}$ implies $(p,q) \in F_{\&}$,
CF6. $(p,q) \xrightarrow{\alpha} R$ and $R \subseteq F_{\&}$ implies $(p,q) \in F_{\&}$.

The *conjunction* $P \sqcap Q$ is obtained from $P \,\&\, Q$ by deleting all states in $F_{\&}$. This deletes all transitions exiting deleted states and removes all deleted states from targets of must-transitions.

Fatal error states are excluded in Rules CF2 through CF5 because we do not care about consistency for these states. Note that the states in E and $F_{\&}$ are different in nature: E-states represent states with possible but unwanted behaviour, whereas $F_{\&}$-states represent contradictory specifications that are impossible to implement.

Figure 3.4 illustrates how conjunction may be employed for perspective-based specification. Consider a double garage for which we want to specify a single controller appropriately operating both garage doors according to an identification of the car requesting passage. We state two requirements for such a controller, each of which may be considered as a separate perspective on the controller:

R_1**:** After a passage request (rqstPass?), the garage shall identify the car (idfy!) and may then open one of the doors (openDoor$_1$!, openDoor$_2$!).

R_2**:** After the car is identified (idfy!), the garage shall open either Door 1 or Door 2 (openDoor$_1$!, openDoor$_2$!).

A representation of these requirements as EMIAs is shown in Figure 3.4. In Specification R_1, the rqstPass?-transition from state 0 to state 1 is the entrance condition that may be triggered by a car in order to request passage. Upon such a request, the garage must identify (idfy!) the car, and may then open Door 1 or Door 2. Requirement R_2 specifies that after an identification, either Door 1 or Door 2 must be opened, i.e., the

choice of door is a result of the identification. The overall specification must satisfy both requirements simultaneously. Hence, we use conjunction in order to construct the greatest lower bound $R := R_1 \sqcap R_2$, which is also shown in Figure 3.4. Notably, the combination of nondeterminism and modalities of action idfy! yields a disjunctive must-transition in the conjunction.

In order to prove that conjunction is the greatest lower bound with respect to the refinement preorder $\sqsubseteq_e$, we need the notion of a witness along the lines of [LV07]:

Definition 3.16 (Witness)**.** Let P and Q be EMIAs with equal alphabets. A set $W \subseteq S_P \times S_Q$ is a *witness* of $P \,\&\, Q$ if, for all $(p, q) \in W$, the following conditions hold:

W1. $p \in E_P$ implies $q \in E_Q \cup U_Q$,
W2. $q \in E_Q$ implies $p \in E_P \cup U_P$,
W3. $p \xrightarrow{o}_P$ implies $q \overset{o}{\rightarrowtail\!\!\!\!\twoheadrightarrow}_Q$ or $q \in E_Q \cup U_Q$,
W4. $q \xrightarrow{o}_Q$ implies $p \overset{o}{\rightarrowtail\!\!\!\!\twoheadrightarrow}_P$ or $p \in E_P \cup U_P$,
W5. $p \xrightarrow{i}_P$ implies $q \overset{i}{\dashrightarrow\!\!\!\!\twoheadrightarrow}_Q$ or $q \in E_Q \cup U_Q$,
W6. $q \xrightarrow{i}_Q$ implies $p \overset{i}{\dashrightarrow\!\!\!\!\twoheadrightarrow}_P$ or $p \in E_P \cup U_P$,
W7. $(p, q) \xrightarrow{\alpha} R'$ implies $R' \cap W \neq \emptyset$.

We instantiate the concept of a witness concretely as follows:

Lemma 3.17 (Concrete Witness)**.** *Let P, Q, R be EMIAs with equal alphabets.*

1. *For any witness W of $P \,\&\, Q$, we have $W \cap F_{\&} = \emptyset$.*
2. *The set $W := \{(p, q) \in S_P \times S_Q \mid \exists r \in S_R.\, r \sqsubseteq_e p$ and $r \sqsubseteq_e q\}$ is a witness of $P \,\&\, Q$.*

Proof. Claim 1 is obvious, so we only prove Claim 2:

W1 By R1, we get $p \in E_P$ implies $r \in E_R$ implies $q \in E_Q \cup U_Q$.

W2 Symmetrically to W1.

W3 If $q \in E_Q$, then W2 applies and there is nothing to show. Otherwise, let $p \xrightarrow{o}_P$. By $r \sqsubseteq_e p$, there is a transition $r \xrightarrow{o}_R$ and, by syntactic consistency and $r \sqsubseteq_e q$, a $q \overset{o}{\rightarrowtail\!\!\!\!\twoheadrightarrow}_Q$.

W4 Symmetrically to W3.

W5 Analogous to W3 when replacing $\xrightarrow{o}$ and $\overset{o}{\rightarrowtail\!\!\!\!\twoheadrightarrow}$ with $\xrightarrow{i}$ and $\overset{i}{\dashrightarrow\!\!\!\!\twoheadrightarrow}$, respectively.

W6 Symmetrically to W5.

W7 Let $(p, q) \in W$ due to r such that $(p, q) \xrightarrow{\omega} R'$ because of C3. By $r \sqsubseteq_e p$, there is a matching $r \overset{\omega}{\rightarrowtail\!\!\!\!\twoheadrightarrow}_R R'$. For all $r' \in R'$, by syntactic consistency, we have a transition $r \overset{\omega}{\rightarrowtail\!\!\!\!\twoheadrightarrow}_R r'$, such that $r \sqsubseteq_e q$ implies the existence of a transition $q \overset{\omega}{\rightarrowtail\!\!\!\!\twoheadrightarrow}_Q q'$ with $r' \sqsubseteq_e q'$. Hence, there is a $(p', q') \in R' \cap W$ due to r'. The case of inputs is shown analogously. □

Next, we show that $\sqcap$ is indeed conjunction:

Proposition 3.18 ($\sqcap$ is And)**.** *If P and Q are EMIAs with equal alphabets, then (i) an R with $R \sqsubseteq_e P$ and $R \sqsubseteq_e Q$ exists iff P and Q are consistent. Further, if P and Q are consistent, then, for any R, (ii) $R \sqsubseteq_e P$ and $R \sqsubseteq_e Q$ iff $R \sqsubseteq_e P \sqcap Q$.*

Proof. (i) "$\Rightarrow$" follows from Lemma 3.17.

(i), (ii) "$\Leftarrow$": Let $R \sqsubseteq_e P \sqcap Q$. We prove that $\mathcal{R} := \{(r,p) \mid \exists q.\, r \sqsubseteq_e p \sqcap q\}$ is an error-preserving modal refinement relation. By choosing $S_R^0 := \{r \in S_R \mid \exists p \sqcap q \in S_{P\sqcap Q}^0.\, (r, p \sqcap q) \in \mathcal{R}\}$ we may conclude (i) "$\Leftarrow$". Let $(r,p) \in \mathcal{R}$ due to q. The proof follows closely the lines of [Buj+16] and proceeds as follows:

R1 If $r \in E_R$, then $p \sqcap q \in E_{P\sqcap Q}$; thus, $p \in E_P$.

R3, R4 Let $p \xrightarrow{\alpha}_P P'$, then we have $q \overset{\alpha}{\rightarrowtail\!\!\!\!\twoheadrightarrow}_Q$ and $p \sqcap q \xrightarrow{\alpha} \{p' \sqcap q' \mid p' \in P', q \overset{\alpha}{\rightarrowtail\!\!\!\!\twoheadrightarrow}_Q q', p' \sqcap q' \text{ defined}\}$. By $r' \sqsubseteq_e p' \sqcap q'$ we get a matching $r \xrightarrow{\alpha}_R R'$, i.e., $\forall r' \in R'\, \exists p' \in P'.\, (r', p') \in \mathcal{R}$. (In case of inputs, $\overset{\alpha}{\rightarrowtail\!\!\!\!\twoheadrightarrow}$ must be replaced by $\overset{\alpha}{\dashrightarrow\!\!\!\!\twoheadrightarrow}$.)

R5, R6 Let $r \overset{\alpha}{\dashrightarrow} r'$. By $r \sqsubseteq_e p \sqcap q$, there is a $p \sqcap q \overset{\alpha}{\rightarrowtail\!\!\!\!\twoheadrightarrow} p' \sqcap q'$ such that $r' \sqsubseteq_e p' \sqcap q'$; thus, $(r', p') \in \mathcal{R}$ due to q'. (In case of inputs, $\overset{\alpha}{\rightarrowtail\!\!\!\!\twoheadrightarrow}$ must be replaced by $\overset{\alpha}{\dashrightarrow\!\!\!\!\twoheadrightarrow}$.)

(ii) "$\Rightarrow$": We show that $\mathcal{R} := \{(r, p \sqcap q) \mid r \sqsubseteq_e p \text{ and } r \sqsubseteq_e q\}$ is an error-preserving modal refinement relation.

R1 Obvious.

R3, R4, R5, R6 As above, the proof closely follows the lines of [Buj+16]. □

As a standard result from category theory, Proposition 3.18 implies that $\sqcap$ is associative:

Corollary 3.19 (Associativity of $\sqcap$)**.** *Conjunction is associative, i.e., for all EMIAs P, Q, and R, we have $P \sqcap (Q \sqcap R) \equiv_e (P \sqcap Q) \sqcap R$.*

3.1.6. Disjunction

It is easy to also define a disjunction operator, which may be employed for specifying alternative implementations:

Definition 3.20 (Disjunction)**.** For a family of EMIAs $\mathcal{P} := (P_j)_{j\in J}$ with equal alphabets, we define the *disjunction* of $\mathcal{P}$ as the following EMIA:

$$\bigsqcup_{j\in J} P_j := \Big(\biguplus_{j\in J} S_{P_j}, I, O, \biguplus_{j\in J} \longrightarrow_{P_j}, \biguplus_{j\in J} \dashrightarrow_{P_j}, \biguplus_{j\in J} S^0_{P_j}, \biguplus_{j\in J} E_{P_j}\Big).$$

In case we consider only two EMIAs P and Q, we write $P \sqcup Q$ for their disjunction.

Proposition 3.21 ($\sqcup$ is Or)**.** *If P_j, for $j \in J$, and R are EMIAs with equal alphabets, then $\bigsqcup_{j\in J} P_j \sqsubseteq_e R$ iff $P_j \sqsubseteq_e R$ for all $j \in J$.*

Proof. Let P_j ($j \in J$) and R be EMIAs with equal alphabets and w.l.o.g. disjoint state sets S_j and S_R, and let $P_j \sqsubseteq_\mathrm{e} R$ due to the error-preserving modal refinement relation $\mathcal{R}_j$. Because, in general, the union of error-preserving modal refinement relations is an error-preserving modal refinement relation, $(\bigcup_{j\in J} \mathcal{R}_j) \cup \mathcal{R}_Q$ is an error-preserving modal refinement relation, too. Vice versa, if $\bigsqcup_{j\in J} P_j \sqsubseteq_\mathrm{e} R$ due to an error-preserving modal refinement relation $\mathcal{R}$, then, for any $j \in J$, $\mathcal{R}_j := \mathcal{R} \cap (S_j \times S_R)$ is a suitable error-preserving modal refinement relation, showing $P_j \sqsubseteq_\mathrm{e} R$. □

It is easy to see that conjunction and disjunction are distributive.

Corollary 3.22 (Distributivity)**.** *Any EMIAs P, Q and R with equal alphabets satisfy the distributive laws:*

1. $(P \sqcap Q) \sqcup R \equiv_\mathrm{e} (P \sqcup R) \sqcap (Q \sqcup R)$,
2. $(P \sqcup Q) \sqcap R \equiv_\mathrm{e} (P \sqcap R) \sqcup (Q \sqcap R)$.

Proof. Obvious, because disjunction of EMIAs is essentially a disjoint union. □

3.1.7. Quotient

The quotient operation is adjoint to parallel composition. It equips the theory with the possibility of component synthesis, which allows for component reuse and incremental, component-based design. Given EMIAs P and D, the quotient of P over D is the coarsest EMIA Q such that the *defining inequality of the quotient*, $Q \otimes D \sqsubseteq_\mathrm{e} P$, holds. We denote the quotient by $P /\!\!/ D$ if it exists. In the following, P is the *dividend* (one may think of it as an overall system specification), D the *divisor* (an already implemented component) and Q the *quotient* (the synthesised completion of D).

We define the quotient for a restricted set of EMIAs, namely where the specification P has no τs and where the divisor D is may-deterministic and without τs. We call D *may-deterministic* if $d \overset{\alpha}{\dashrightarrow} d'$ and $d \overset{\alpha}{\dashrightarrow} d''$ implies $d' = d''$ for all d, d', d'' and α. Due to syntactic consistency, a may-deterministic EMIA has no disjunctive must-transitions, i.e., the target sets of must-transitions are singletons. We discuss the difficulties with nondeterministic divisors in Section 3.4.2.

Like conjunction above, we define the quotient in two stages. Regarding the choice of the input and output alphabets in the following definition, we adopt the one by Chilton [Chi13] and Raclet et al. [Rac+11]. Alternative choices are discussed in Section 3.4.2.

Definition 3.23 (Pre-quotient)**.** Let P and D be τ-free EMIAs with $A_D \subseteq A_P$ and $O_D \subseteq O_P$. The *pre-quotient of P over D* is defined as the EMIA $P \oslash D := (S_P \times S_D \cup \{\top\}, I, O, \longrightarrow, \dashrightarrow, S_P^0 \times S_D^0, E, U)$, where $I := I_P \cup O_D$, $O := O_P \setminus O_D$, $E := E_P \times (S_D \setminus E_D)$ and $U := (U_P \times S_D) \cup (E_P \times E_D) \cup \{\top\}$. The transition relations of a state (p, d) are defined by the following rules:

PQ1. $(p,d) \xrightarrow{a} P' \times \{d\}$ if $p \xrightarrow{a} P'$ and $a \notin A_D$,
PQ2. $(p,d) \xrightarrow{a} P' \times D'$ if $p \xrightarrow{a} P'$ and $d \xrightarrow{a} D'$,

PQ3. $(p,d) \dashrightarrow^{a} (p',d)$ if $p \dashrightarrow^{a}$ and $a \notin A_D$,
PQ4. $(p,d) \dashrightarrow^{a} (p',d')$ if $p \dashrightarrow^{a}$ and $d \dashrightarrow^{a} d'$,
PQ5. $(p,d) \dashrightarrow^{a} \top$ if $p \not\dashrightarrow^{a}$ and $d \not\dashrightarrow^{a}$.

A state $q = (p,d)$ in $P \oslash D$ encodes the condition that q should be the coarsest state with respect to $\sqsubseteq_{\mathrm{e}}$ such that q composed in parallel with d refines p. The purpose of the new state $\top$ is to ensure that U is nonempty, in order to have a universal target state in Rule PQ5. In case U is nonempty anyway, an arbitrary state from U may replace $\top$. With this in mind, we now justify the choices of E and U and the rules of Definition 3.23 intuitively. A formal proof is given in Lemma 3.26 and Theorem 3.27 below.

An error state $q \in E$ of the quotient satisfies $q \otimes d \sqsubseteq_{\mathrm{e}} p$ for some state $d \in S_D$ if and only if $p \in E_P \cup U_P$. However, if $d \in E_D$ or $p \in U_P$, then nothing is required for q to satisfy $q \otimes d \sqsubseteq_{\mathrm{e}} p$ and, hence, q has to be universal instead of erroneous in order to ensure the maximality of the quotient. This justifies the choices of E and U.

Rule PQ1 is necessary due to the following consideration. If P has an a-must-transition where a is unknown to D, then this can only originate from an a-must-transition in the quotient Q that we wish to construct. To be most permissive, each $p' \in P'$ must have a match in $Q \otimes D$. The corresponding consideration is true for Rule PQ3, which also ensures syntactic consistency for Rule PQ1.

Rule PQ2 is obvious in the light of the choice of alphabet in Definition 3.23. Because $P \oslash D$ has all actions of P and D in its alphabet, it also needs an a-must-transition to produce such a transition at $(p,d) \otimes d$. Here, Rule PQ4 is the companion rule for guaranteeing syntactic consistency.

Rule PQ5 makes $P \oslash D$ as coarse as possible. The input a-may-transitions introduced here just disappear in $(P \oslash D) \otimes D$ since a is blocked by D.

It is easy to see that $P \oslash D$ is indeed an EMIA. Up to now we have only defined the pre-quotient. Considering a candidate pair (p,d), it may be impossible that p is refined by a state resulting from a parallel composition with d; this depends, e.g., on the modalities and the labels of the transitions leaving p and d. We call such pairs *divisionally inconsistent states* and remove them from the pre-quotient. For example, consider states $p \xrightarrow{a}$ and $d \dashrightarrow^{a}$ such that $d \not\xrightarrow{a}$; no parallel composition with d refines p. While may-transitions can be refined by removing them and disjunctive transitions can be refined to subsets of their targets in order to prevent the reachability of inconsistent states, all states having a must-transition to only inconsistent states must also be removed.

Definition 3.24 (Quotient)**.** Let $P \oslash D$ be the pre-quotient of P over D. The set $F_\oslash \subseteq S_P \times S_D$ of *divisionally inconsistent states* is defined as the least set satisfying the following rules:

QF1. $p \notin U_P \cup E_P$ and $d \in U_D \cup E_D$ implies $(p,d) \in F_\oslash$,
QF2. $p \xrightarrow{a}_P$ and $d \not\xrightarrow{a}_D$ and $a \in A_D$ implies $(p,d) \in F_\oslash$,
QF3. $(p,d) \xrightarrow{a}_{P \oslash D} R'$ and $R' \subseteq F_\oslash$ implies $(p,d) \in F_\oslash$.

The *quotient* $P /\!\!/ D$ is obtained from $P \oslash D$ by deleting all states in $F_\oslash$. This also removes any may- or must-transition exiting a deleted state and any may-transition

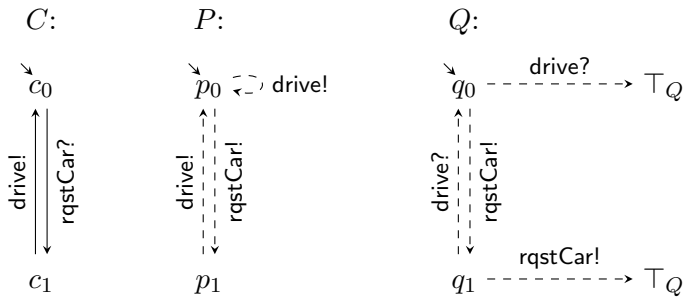


Figure 3.5.: Synthesis of a user interface Q from a given component C and a global specification P, where $A_C := \{\mathsf{rqstCar?}\}/\{\mathsf{drive!}\}$, $A_P := \emptyset/\{\mathsf{rqstCar!}, \mathsf{drive!}\}$ and $A_Q := \{\mathsf{drive?}\}/\{\mathsf{rqstCar!}\}$.

entering a deleted state; in addition, deleted states are removed from targets of disjunctive must-transitions. If $(p, d) \in S_{P/\!\!/D}$, then we write $p \mathbin{/\!\!/} d$. If $S^0_{P/\!\!/D}$ is empty, then the quotient of P over D is inconsistent.

Rule QF1 captures the division by universal states and error states. A state $d \in U_D \cup E_D$ in parallel with any state is either universal or an error state, and does not refine $p \notin U_P \cup E_P$. Rule QF2 is obvious since (p, d) cannot ensure that $p \xrightarrow{a}_P$ is matched if d has no a-must-transition, as an a-may-transition or a forbidden action a at d can in no case compose to a refinement of a must-transition at p. Rule QF3 propagates back all inconsistencies that cannot be avoided by refining.

Since $P \oslash D$ is an EMIA and since syntactic consistency and the special states are preserved by pruning, $P \mathbin{/\!\!/} D$ is an EMIA, too. If the target set of a disjunctive must-transition became empty due to pruning, i.e., $R' \subseteq F_\oslash$, Rule QF3 would be applicable and the source state and its must-transition are deleted.

Example 3.25. We reconsider our running example of a driving assistant system. Figure 3.5 shows a simple implementation C of a car that allows a user to request assistance (rqstCar?) upon which the car drives into or out of a garage autonomously (drive!). We consider C as a given implementation which we want to reuse in order to synthesise a specification of the user interface. The composition $C \otimes U$ of the car C and its user interface U must satisfy the global specification P which requires that, after some request, the car may drive and new requests are blocked until the drive is completed. A specification Q of the user interface may now be synthesised from P and C by quotienting, i.e., $Q := P \mathbin{/\!\!/} C$. Note that drive? is an input action in Q. The two transitions leading to universal states (drive? in q_0 and rqstCar! in q_1) are only due to the maximality of Q. They disappear in the parallel composition with C. It is easy to see that the defining inequality $Q \otimes C \sqsubseteq_\mathrm{e} P$ is satisfied. The example also shows that, in general, we do not have equality of $(P \mathbin{/\!\!/} C) \otimes C$ and P.

We show next that the quotient operation above yields the coarsest EMIA satisfying the defining inequality. For this proof, the following lemma ensures that errors and inconsistencies of $/\!\!/$ are preserved across refinement:

Lemma 3.26. *Let P, D and Q be EMIAs such that P is τ-free, D is τ-free and may-deterministic, $A_D \subseteq A_P$, $O_D \subseteq O_P$, $O_Q = O_P \setminus O_D$ and $I_Q = I_P \cup O_D$. Further, let p, d, q be states in P, D, Q, respectively. Then, the following statements hold:*

1. *If $q \otimes d \sqsubseteq_{\mathrm{e}} p$, then $(p, d) \notin F_{\oslash}$.*
2. *If $q \sqsubseteq_{\mathrm{e}} p /\!\!/ d$ and $p \notin U_P \cup E_P$, then $q \otimes d \notin E_{Q \otimes D}$.*

Proof. We write $\longrightarrow_{\otimes}$, $\longrightarrow_{\oslash}$ and $\longrightarrow_{/\!/}$ as shorthands for $\longrightarrow_{Q\otimes D}$, $\longrightarrow_{P \oslash D}$ and $\longrightarrow_{P /\!/ D}$, respectively, and analogously for may-transitions.

Claim 1: We show that $(q \otimes d \sqsubseteq_{\mathrm{e}} p) \wedge ((p, d) \in F_{\oslash})$ implies a contradiction. We prove this by induction on the rules of Definition 3.24, where our induction hypothesis is formalised as $H(p, d) \equiv \forall q.\, (q \otimes d \sqsubseteq_{\mathrm{e}} p \wedge (p, d) \in F_{\oslash}) \implies \bot$.

QF1 $p \notin E_P \cup U_P$ and $d \in E_D \cup U_D$: By Definition 3.7, we have $q \otimes d \in E_{Q\otimes D} \cup U_{Q\otimes D}$, and $q \otimes d \sqsubseteq_{\mathrm{e}} p$ implies $p \in E_P \cup U_P$ which contradicts our assumption.

QF2 $p \xrightarrow{a}$, $d \not\xrightarrow{a}$ and $a \in A_D$: By $q \otimes d \sqsubseteq_{\mathrm{e}} p$, we have $q \otimes d \xrightarrow{a}_{\otimes}$, which can only be due to P2 or P3; thus, $d \xrightarrow{a}$, which is a contradiction.

QF3 $(p, d) \xrightarrow{a}_{\oslash} R'$ with $R' \subseteq F_{\oslash}$: By induction hypothesis $H(p', d')$ holds for all $(p', d') \in R'$. The transition is due to one of the rules PQ1 and PQ2:

- **PQ1** $p \xrightarrow{a} P'$, $a \notin A_D$ and $R' = P' \times \{d\}$: By $q \otimes d \sqsubseteq_{\mathrm{e}} p$, we have $q \otimes d \xrightarrow{a}_{\otimes} Q' \times \{d\}$ for some Q' such that $\forall q' \in Q'.\, \exists p' \in P'.\, q' \otimes d \sqsubseteq_{\mathrm{e}} p'$. Since $(p', d) \in R' \subseteq F_{\oslash}$, $H(p', d)$ implies a contradiction.
- **PQ2** $p \xrightarrow{a} P'$, $d \xrightarrow{a} \{d'\}$ and $R' = P' \times \{d'\}$: By $q \otimes d \sqsubseteq_{\mathrm{e}} p$, there is a Q' with $q \xrightarrow{a} Q'$ and $\forall q' \in Q'.\, \exists p' \in P'.\, q' \otimes d' \sqsubseteq_{\mathrm{e}} p'$. Due to $(p', d') \in R' \subseteq F_{\oslash}$ we can derive a contradiction from $H(p', d')$.

Claim 2: We show that $(q \sqsubseteq_{\mathrm{e}} p /\!\!/ d) \wedge (p \notin E_P) \wedge (q, d) \in E_{Q\otimes D}$ implies a contradiction. By Definition 3.7, there are two cases for $(q, d) \in E_{Q\otimes D}$:

A $(q, d) \in E_Q \times S_D$: By $q \sqsubseteq_{\mathrm{e}} p /\!\!/ d$, we have $p /\!\!/ d \in E_{P/\!\!/D}$. Hence, $p \in E_P$, which is a contradiction.

B $(q, d) \in S_Q \times E_D$: By QF1, $(p, d) \in F_{\oslash}$, which contradicts $q \sqsubseteq_{\mathrm{e}} p /\!\!/ d$. □

Now, we can show that $/\!\!/$ is indeed a quotient operator with respect to $\otimes$:

Theorem 3.27 ($/\!\!/$ is a Quotient Operator with respect to $\otimes$)**.** *Let P, D and Q be EMIAs such that P is τ-free, D is τ-free and may-deterministic, $A_D \subseteq A_P$, $O_D \subseteq O_P$, $O_Q = O_P \setminus O_D$ and $I_Q = I_P \cup O_D$. Then, $Q \sqsubseteq_{\mathrm{e}} P /\!\!/ D$ iff $Q \otimes D \sqsubseteq_{\mathrm{e}} P$.*

Proof. We use the same shorthands as in Lemma 3.26.

"⇒": We show that $\mathcal{R} := \{(q \otimes d, p) \in S_{Q\otimes D} \times S_P \mid q \sqsubseteq_{\mathrm{e}} p /\!\!/ d \text{ or } p \in U_P\}$ is an error-preserving modal refinement relation. We only have to consider a $(q \otimes d, p) \in \mathcal{R}$ with $p \notin U_P$. Note that Cases R4 and R6 are mostly analogous to Cases R3 and R5, respectively.

R1 $q \otimes d \notin E_{Q\otimes D}$ iff (by Definition 3.7) $q \notin E_Q \wedge d \notin E_D$ iff (by $q \sqsubseteq_e p /\!\!/ d$) $p /\!\!/ d \notin E_{P/\!\!/D} \wedge d \notin E_D$. Definition 3.23 implies $p \notin E_P$. Vice versa, $p \notin E_P$ implies $p /\!\!/ d \notin E_{P/\!\!/D}$. Now, $q \sqsubseteq_e p /\!\!/ d$ implies that $p /\!\!/ d$ is consistent, hence, $d \notin E_D$ and, due to Definition 3.7, $q \otimes d \notin E_{Q\otimes D}$.

R2 By Definition 3.23, $p \notin U_P$ implies $p /\!\!/ d \notin U_{P/\!\!/D}$. Due to $q \sqsubseteq_e p /\!\!/ d$, we have $q \notin U_Q$. Now, QF1 implies $d \notin U_D$, hence, $q \otimes d \notin U_{Q\otimes D}$.

R3 $p \xrightarrow{i} P'$ for $i \in I_P$:

1. If $i \in A_D$ and $d \xrightarrow{i} \{d'\}$, then PQ2 implies $(p,d) \xrightarrow{i}_{\oslash} P' \times \{d'\}$. In $P /\!\!/ D$, the target set might only be a subset $P'' \times \{d'\}$ of $P' \times \{d'\}$. By $q \sqsubseteq_e p /\!\!/ d$, we have $q \xrightarrow{i} Q'$ for some Q' such that $\forall q' \in Q'.\, \exists p' \in P''.\, q' \sqsubseteq_e p' /\!\!/ d'$, whence $(q' \otimes d', p') \in \mathcal{R}$. Now, by P3, there is a transition $(q,d) \xrightarrow{i}_{\otimes} Q' \times \{d'\}$.
2. If $i \in A_D$ and $d \not\xrightarrow{i}$, then $(p,d) \in F_{\oslash}$ by QF2, which is impossible since $p /\!\!/ d$ is consistent.
3. If $i \notin A_D$, the proof is analogous to Case 1 with $d = d'$, when replacing PQ2 by PQ1 and P3 by P1.

R4 $p \xrightarrow{o} P'$ for $o \in O_P$: Here, the same arguments as for R3 apply.

R5 $q \otimes d \dashrightarrow^{i}_{\otimes} q' \otimes d'$ and $i \in I_P = I_{Q\otimes D}$: This transition is due to one of the rules P4 or P6. Rule P5 is impossible as $A_Q = A_P \supseteq A_D$.

P4 $q \dashrightarrow^{i} q'$ and $i \notin A_D$: We have $d = d'$, and $q \sqsubseteq_e p /\!\!/ d$ implies $p /\!\!/ d \dashrightarrow^{i}_{/\!\!/} p' /\!\!/ d''$ for some p', d'' such that $q' \sqsubseteq_e p' /\!\!/ d''$. Since $i \notin A_D$, we get $d = d''$ and $p \dashrightarrow^{i} p'$ by PQ3. We have $(q' \otimes d', p') \in \mathcal{R}$ since $q' \sqsubseteq_e p' /\!\!/ d'$.

P6 $q \dashrightarrow^{i} q'$ and $d \dashrightarrow^{i} d'$: Since $q \sqsubseteq_e p /\!\!/ d$, we conclude $p /\!\!/ d \dashrightarrow^{i}_{/\!\!/} p' /\!\!/ d''$ for some p', d'' with $q' \sqsubseteq_e p' /\!\!/ d''$. This can be due to PQ3 or PQ4; in both cases we have $p \dashrightarrow^{i} p'$. Due to may-determinism, $d'' = d'$ and, since $q' \sqsubseteq_e p' /\!\!/ d'$, we have $(q' \otimes d', p') \in \mathcal{R}$.

R6 $q \otimes d \dashrightarrow^{o}_{\otimes}$ and $o \in O_P = O_{Q\otimes D}$: The proof proceeds analogous to the one of R5.

"⇐": We show that $\mathcal{R} := \{(q, p /\!\!/ d) \in Q \times (P /\!\!/ D) \mid q \otimes d \sqsubseteq_e p \text{ or } p /\!\!/ d \in U_{P/\!\!/D}\}$ is an error-preserving modal refinement relation. It suffices to consider some $(q, p/\!\!/d) \in \mathcal{R}$ with $p /\!\!/ d \notin U_{P/\!\!/D}$.

R1 $q \in E_Q$ implies (by Definition 3.7) $q \otimes d \in E_{Q\otimes D}$ iff (by $q \otimes d \sqsubseteq_e p$) $p \in E_P$ iff (by Definition 3.23) $p /\!\!/ d \in E_{P\oslash D}$. For the reverse direction, it remains for us to show that the first implication can be reversed, i.e., that $d \notin E_D$. By $q \otimes d \in E_{Q\otimes D}$ and $q \otimes d \sqsubseteq_e p$, we have $p \in E_P \cup U_P$. Hence, $p /\!\!/ d \notin U_{P/\!\!/D}$ implies $d \notin E_D$.

R2 By Definition 3.23, $p /\!\!/ d \notin U_{P/\!\!/D}$ implies $p \notin U_P$ and $(p,d) \notin E_P \times E_D$. There are two cases:

1. $p \in E_P$ and $d \notin E_D$: $q \otimes d \sqsubseteq_{\mathrm{e}} p$ implies $q \otimes d \in E_{Q \otimes D}$; hence, $q \in E_Q$.
2. $p \notin E_P$: $q \otimes d \sqsubseteq_{\mathrm{e}} p$ implies $q \otimes d \notin E_{Q \otimes D} \cup U_{Q \otimes D}$.

In both cases we conclude $q \notin U_Q$.

R3 $p /\!\!/ d \xrightarrow{i}_{/\!\!/} R' \subseteq P' \times \{d'\}$ for $i \in I_{P/\!\!/D}$, where $(p,d) \xrightarrow{i}_{\oslash} P' \times \{d'\}$ is due to one of the rules PQ1 or PQ2, and R' consists of the consistent states of $P' \times \{d'\}$. In the following, we use $A_P = A_Q$ throughout.

PQ1 $p \xrightarrow{i} P'$, $d = d'$ and $i \notin A_D$: By $q \otimes d \sqsubseteq_{\mathrm{e}} p$, we have a transition $q \otimes d \xrightarrow{i}_{\otimes} Q' \times \{d''\}$ for some Q', d'' with $\forall q' \in Q'.\, \exists p' \in P'.\, q' \otimes d'' \sqsubseteq_{\mathrm{e}} p'$. Since $i \notin A_D$, this transition can only be due to Rule P1; hence, $q \xrightarrow{i} Q'$ and $d'' = d$. By Lemma 3.26, $q' \otimes d \sqsubseteq_{\mathrm{e}} p'$ implies $p' /\!\!/ d \notin F_{\oslash}$; thus, $p' /\!\!/ d \in R'$.

PQ2 $p \xrightarrow{i} P'$ and $d \xrightarrow{i} d'$: By $q \otimes d \sqsubseteq_{\mathrm{e}} p$, we get $q \otimes d \xrightarrow{i}_{\otimes} Q' \times \{d'\}$ for some Q' such that $\forall q' \in Q'.\, \exists p' \in P'.\, q' \otimes d' \sqsubseteq_{\mathrm{e}} p'$. The transition must result from P3, and the rest of the proof is as in PQ1.

R4 $p /\!\!/ d \xrightarrow{o}_{/\!\!/} R'$ with $o \in O_{P/\!\!/D} = O_P \setminus O_D$: The same arguments as for R3 apply.

R5 $q \dashrightarrow^{i} q'$ for $i \in I_Q$:

1. $i \notin A_D$: By P4, we have $(q,d) \dashrightarrow^{i}_{\otimes} (q',d)$. There is a transition $p \dashrightarrow^{i} p'$ for some p' with $q' \otimes d \sqsubseteq_{\mathrm{e}} p'$, because of $q \otimes d \sqsubseteq_{\mathrm{e}} p$. By PQ3, we have $(p,d) \dashrightarrow^{i}_{\oslash} (p',d)$, and Lemma 3.26 implies the consistency of $p' /\!\!/ d$; hence, $p /\!\!/ d \dashrightarrow^{i}_{/\!\!/} p' /\!\!/ d$.
2. $i \in A_D$ and $d \not\dashrightarrow^{i}$: Due to $d \not\xrightarrow{i}_D$ and QF2, we have $p \not\xrightarrow{i}_P$. Hence, PQ5 yields $p /\!\!/ d \dashrightarrow^{i} \top$, and $(q', \top) \in \mathcal{R}$ is trivial.
3. $i \in A_D$ and $d \dashrightarrow^{i} d'$: By P6, a transition $(q,d) \dashrightarrow^{i}_{\otimes} (q',d')$ exists. The proof proceeds as for Case 1, except for using PQ4 instead of PQ3.

R6 $q \dashrightarrow^{o} q'$ for $o \in O_Q$:

1. $o \in A_D$, $d \dashrightarrow^{o} d'$ for some d': By P6, we have $(q,d) \dashrightarrow^{o}_{\otimes} (q',d')$ and, by $q \otimes d \sqsubseteq_{\mathrm{e}} p$, we obtain $p \dashrightarrow^{o} p'$ for some p' with $q' \otimes d' \sqsubseteq_{\mathrm{e}} p'$. Applying PQ4, we get $(p,d) \dashrightarrow^{o}_{\oslash} (p',d')$. Lemma 3.26 implies the consistency of $p' /\!\!/ d'$; hence, $p /\!\!/ d \dashrightarrow^{o}_{/\!\!/} p' /\!\!/ d'$.
2. $o \in A_D$, $d \not\dashrightarrow^{o}$: Analogous to case R5(2).
3. $o \notin A_D$: $q \otimes d \dashrightarrow^{o}_{\otimes} q' \otimes d$ by P4. Due to $q \otimes d \sqsubseteq_{\mathrm{e}} p$, there is a $p \dashrightarrow^{o} p'$ for some p' with $q' \otimes d \sqsubseteq_{\mathrm{e}} p'$. The rest follows as in the proof of Case 1, applying PQ3 instead of PQ4. □

This theorem establishes $/\!\!/$ as an adjoint of $\otimes$ when setting $D \triangleright P := P /\!\!/ D$ (see Section 2.1.1). Due to the associativity and the compositionality of $\otimes$, Lemma 2.1 holds for $/\!\!/$. In particular, we may conclude that $/\!\!/$ is monotonic with respect to $\sqsubseteq_e$ in the left argument and antitonic with respect to the right argument, and that $/\!\!/$ satisfies a De Morgan-like law.

Corollary 3.28 (Monotonicity to the Left of $/\!\!/$ with respect to $\sqsubseteq_e$)**.** *Let P_1, P_2, D be EMIAs with $P_1 \sqsubseteq_e P_2$. If P_1, P_2 are τ-free and D is τ-free and may-deterministic, then $P_1 /\!\!/ D \sqsubseteq_e P_2 /\!\!/ D$.*

Corollary 3.29 (Antitonicity to the Right of $/\!\!/$ with respect to $\sqsubseteq_e$)**.** *Let P, D_1, D_2 be EMIAs with $D_1 \sqsubseteq_e D_2$. If P is τ-free and D_1, D_2 are τ-free and may-deterministic, then $P /\!\!/ D_1 \sqsupseteq_e P /\!\!/ D_2$.*

Corollary 3.30 (De Morgan-like Law for $/\!\!/$)**.** *Let P, Q and R be EMIAs, then $P /\!\!/ (Q \sqcup R) \sqsubseteq_e (P /\!\!/ Q) \sqcap (P /\!\!/ R)$.*

By Lemma 2.1 and commutativity of parallel composition, we get:

Corollary 3.31 (Quotienting by Composition)**.** *Let P, Q and R be EMIAs, then $P /\!\!/ (Q \otimes R) \equiv_e (P /\!\!/ Q) /\!\!/ R$.*

3.1.8. Implication and Negation

In addition to conjunction and disjunction, we would appreciate to define further logical operators like implication and negation. Implication $\rightarrow$, as an adjoint to conjunction (see Section 2.1.1), is defined by the condition $X \sqsubseteq_e P \rightarrow C$ iff $X \sqcap P \sqsubseteq_e C$. In particular, we have $P \sqsubseteq_e C$ iff $P \rightarrow C \equiv_e \top$. Negation arises as the special case $\neg P := P \rightarrow \bot$. A straightforward way of defining implication is by setting $P \rightarrow C := \bigsqcup\{X \mid X \sqcap P \sqsubseteq_e C\}$. However, this declarative definition is impractical due to the infinite disjunction. Unfortunately, we can show that DMTS and, thus, any MTS-based interface theory is not closed under negation, so that an operational construction of implication and negation in the spirit of the other operators is impossible (see Theorem 3.32).

In a trace-based setting similar to deterministic IA, Dill argues that safety properties are not closed under negation and, therefore, a negation operator does not exist in his setting [Dil89]. However, it is unclear in how far this argument applies to an MTS-based setting where must-transitions express a limited form of liveness.

Gössler and Raclet [GR09] introduced an underapproximation $\rightsquigarrow$ of implication for deterministic MTS. This *sub-implication* satisfies $X \sqsubseteq P \rightsquigarrow C \implies X \sqcap P \sqsubseteq C$, but the reverse direction does not hold in general. Specifications X and P are called *non-conflicting*—a concept introduced as *independence* in [LSW95]—if $F_{\&} = \emptyset$. For non-conflicting specifications X and P, Gössler and Raclet show that the reverse direction also holds, i.e., $X \sqcap P \sqsubseteq C \implies X \sqsubseteq P \rightsquigarrow C$. In particular, $P \rightsquigarrow C$ and P are non-conflicting. However, Gössler and Raclet do not consider that non-conflicting X and P satisfy $X \sqcap P \sqsubseteq C$ only if P and C, as well as X and C, are also non-conflicting. This undermines the purpose of disjunctive must-transitions to provide a choice between

alternatives in case of a conflict such that, in a non-conflicting conjunction, all alternatives must be preserved. Further, as $P \rightsquigarrow \bot$ and P are non-conflicting, $P \rightsquigarrow \bot \equiv_{\sqsubseteq} \bot$ for all consistent specifications P, which renders negation completely useless. These issues significantly restrict the usability of sub-implication.

Gössler and Raclet also show that an implication operator does not exist for MTS. Because their counter-example does not work in the nondeterministic setting of EMIA, we provide a more general argument:

Theorem 3.32 (DMTS and Negation)**.** *Disjunctive Modal Transition Systems (DMTS) are not closed under negation; hence, conjunction has no adjoint in DMTS.*

Proof. DMTS have been shown to be equally expressive than Hennessy-Milner-Logic with greatest fixed points (νHML), if the number of initial states is required to be finite [Ben+13]. If DMTS were closed under negation, then least fixed points were expressible by defining $\mu X.\, \phi(X) := \neg\nu X.\, \neg\phi(\neg X)$, and DMTS would be equally expressive to the modal μ-calculus, which is strictly more expressive that νHML. □

A concrete example that illustrates the difficulty with negation is a specification S over alphabet $\{a\}$ with a single state s and a looping transition $s \xrightarrow{a} s$. The negation of S comprises all implementations that are inconsistent with S, i.e., all implementations that include a finite chain of a-transitions, e.g., $T\colon t_0 \xrightarrow{a} t_1 \xrightarrow{a} t_2$. In some sense, sub-implication captures the non-negative part of implication, which explains the relation to non-conflicting conjunctions.

In order to extend the proposed sub-implication to nondeterministic interfaces, one may employ an exponential construction similar to the one presented in [Ben+13] for the quotient. However, this would lead to similar complications (see also Section 3.1.7) and would still result in a questionable underapproximation.

3.1.9. Alphabet Extension

In perspective-based specification as employed in software engineering, one wishes to specify a component from multiple separate perspectives (also called views). Each perspective should be specifiable independently of the other perspectives and consider only those actions that are relevant for the current perspective, i.e., each perspective has its own alphabet; these alphabets may be identical, disjoint or overlapping. The specification of the overall component should then arise as the conjunction of all perspective specifications.

So far, conjunction, disjunction and refinement are defined for EMIAs P and Q with equal alphabets, i.e., where $I_P = I_Q$ and $O_P = O_Q$. That is, given two EMIAs specifying perspectives with different alphabets, one wishes to merge these alphabets in order to make conjunction, disjunction and refinement applicable. This may be done by employing alphabet extension operators. In addition, alphabet extension may be employed to add extra features that are not covered by the specification interface (see [Rac+11]).

Definition 3.33 (Universal Alphabet Extension)**.** Given an EMIA $P := (S_P, I_P, O_P, \longrightarrow_P, \dashrightarrow_P, S_P^0, E_P, U_P)$, a state $\top \notin S_P$ and two disjoint sets I and O of input and

respectively output actions with $I \cap A_P = \emptyset = O \cap A_P$, the *universal alphabet extension* of P by I and O is the EMIA $P' := (S_P \cup \{\top\}, I_P \cup I, O_P \cup O, \longrightarrow, \dashrightarrow, S_P^0, E_P, U_P \cup \{\top\})$ obtained from P by adding to each state $s \in S_P \setminus (E_P \cup U_P)$ transitions $s \overset{a}{\dashrightarrow} \top$ for all $a \in I \setminus I_P$ and all $a \in O \setminus O_P$.

Definition 3.34 (Alphabet Extension Operators)**.** An *alphabet extension operator* over a pair of disjoint sets I and O is a function ϵ on EMIA that maps each EMIA P to a refinement of its universal extension P'. Operator ϵ is *compositional* if the preorder given by $P \sqsubseteq_\epsilon Q$ iff $P \sqsubseteq \epsilon(Q)$ is compositional:

Following the lines of MI and MIA, the operations on EMIAs can be lifted to different alphabets by extending the alphabets of the operands by their mutually foreign actions. When a specification's alphabet is extended, the least possible assumptions should be made on a new action a, while the same specification with respect to known actions should hold before and after a. This can be achieved by adding an optional a-loop to each state. For output actions this is straightforward, but the exact meaning of optional input transitions depends on the desired composition concept (see Section 3.3, Issue I4). Here, we define the standard error-aware alphabet extension employed in interface theories:

Definition 3.35 (Error-aware Alphabet Extension & Refinement)**.** Given an EMIA $P := (S_P, I_P, O_P, \longrightarrow, \dashrightarrow, S_P^0, E_P, U_P)$, a state $\mathsf{err} \in E_P$ and disjoint action sets I and O satisfying $I \cap A_P = \emptyset = O \cap A_P$, the *error-aware alphabet extension of P by I and O* is given by $[P]_{I,O} := (S_P, I_P \cup I, O_P \cup O, \longrightarrow', \dashrightarrow', S_P^0, E_P, U_P)$ for $\longrightarrow' := \longrightarrow \cup \{(p, i, \{p, \mathsf{err}\}) \mid p \in S_P \setminus (E_P \cup U_P), i \in I\}$ and $\dashrightarrow' := \dashrightarrow \cup \{(p, i, \mathsf{err}) \mid p \in S_P \setminus (E_P \cup U_P), i \in I\} \cup \{(p, a, p) \mid p \in S_P \setminus (E_P \cup U_P), a \in I \cup O\}\}$. We often write $[p]_{I,O}$ for p as state of $[P]_{I,O}$, or $[p]$ in case I, O are understood from the context.

For EMIAs P and Q with $p \in P$, $q \in Q$, $I_P \supseteq I_Q$ and $O_P \supseteq O_Q$, we define $p \sqsubseteq_{\mathrm{e}}' q$ if $p \sqsubseteq_{\mathrm{e}} [q]_{I_P \setminus I_Q, O_P \setminus O_Q}$. Since $\sqsubseteq_{\mathrm{e}}'$ extends $\sqsubseteq_{\mathrm{e}}$ to EMIAs with different alphabets, we write $\sqsubseteq_{\mathrm{e}}$ for $\sqsubseteq_{\mathrm{e}}'$ and abbreviate $[q]_{I_P \setminus I_Q, O_P \setminus O_Q}$ by $[q]_P$; the same notations are used for P and Q.

Lemma 3.36. *Operator $[\cdot]$ of Definition 3.35 is a compositional alphabet extension operator in the sense of Definition 3.34.*

Proof. It is obvious that $[\cdot]$ is an alphabet extension operator. The proof of its compositionality is standard and may be found, e.g., in [Buj+16]. □

A separate alphabet extension operator may be defined for unanimous and broadcast parallel composition. A broadcasting alphabet extension operator is similar to the error-aware one, except that we do not add err as a target of input transitions. A unanimous extension operator simply adds may loops to all states for all new actions. Alternatively, a mixed extension combining different composition concepts for each state and each new action is also possible.

3.2. Error-abstracting Modal Interface Automata

Because IA-based interface theories prune errors, it is important to investigate the relation between such error-abstracting interface theories and our error-preserving EMIA theory. We do this for MIA3 [Buj+16] because it was (before EMIA) the most general IA-based interface theory to date in that it is nondeterministic rather than deterministic and optimistic rather than pessimistic, thus subsuming MI [Rac+11] and MIO [Bau+10] (with respect to strong compatibility). In this section, we establish a Galois insertion between MIA and a subtheory of EMIA, i.e., a Galois connection (γ, α) for which $\alpha \circ \gamma = \mathrm{id}_{\mathsf{MIA}}$ up to mutual refinement $\equiv_{\mathrm{m}}$ of MIAs [CC77]. Recall that states from which a communication mismatch is reachable via output- or τ-transitions are called illegal. Intuitively, α abstracts from EMIAs by considering all illegal states to be equivalent, and γ concretises MIAs as EMIAs without any loss of information.

3.2.1. Error-abstraction

We slightly generalise the MIA3 theory given in [Buj+16; FL16b] in order to get a more uniform presentation of MIA and EMIA, which also simplifies the proof of Theorem 3.48 compared to [FL16b] by avoiding infinite disjunctions.

Definition 3.37 (Error-abstracting Modal Interface Automata [Buj+16])**.** An EMIA P is called an *error-abstracting Modal Interface Automaton* (MIA) if

M1.	$\forall p \in S_P \backslash (E_P \cup U_P), i \in I_P. \exists p' \in S_P. p \dashrightarrow^{i} p'$,	(input enabledness)
M2.	$p \dashrightarrow^{i} p' \implies \exists P'. p' \in P' \wedge p \xrightarrow{i} P'$,	(input must)
M3.	$s \in E_P \cup U_P \wedge p \dashrightarrow^{a} s \implies a \in I_P$.	(error abstraction)

We write EMIA' for the class of EMIAs satisfying M1 and M2, and MIA for the class of MIAs. With $\sqsubseteq_{\mathrm{m}}$ we denote the restriction of $\sqsubseteq_{\mathrm{e}}$ to MIAs.

Note that input enabledness and the input must condition do not restrict our definition of MIA when compared to MIA3 because a transition may target states in E_P and U_P. The purpose of these conditions is to distinguish the error-aware parallel composition of MIA from the unanimous parallel composition of MTS in our unified model, because the latter mode of composition is not supported by MIA. The intuition behind the distinguishing axiom M3 is that a MIA is error-abstracted by definition, i.e., no error state may be reached via output- or τ-transitions. We generalise MIA3 by permitting that an action $a?$ chooses nondeterministically between an error state and a regular successor state, e.g., $p' \xleftarrow{a?} p \xrightarrow{a?} \mathsf{err}$. It is easy to see that EMIA' is closed under $\otimes$:

Lemma 3.38. *If $P, Q \in \mathsf{EMIA}'$, then $P \otimes Q \in \mathsf{EMIA}'$.*

Proof. As a direct consequence of Definition 3.7, if $p \in S_P$ and $q \in S_Q$ are input enabled and satisfy the input must condition, then $p \otimes q$ satisfies both conditions, too. □

In order to make parallel composition on MIA respect error abstraction, we need to consider the reachability of illegal states:

Definition 3.39 (Backward Closure)**.** Let P be an EMIA, $X \subseteq A_P \cup \{\tau\}$ and $S \subseteq S_P$. The *X-backward closure of S in P* is the smallest set $\mathrm{bcl}_P^X(S) \subseteq S_P$ such that $S \subseteq \mathrm{bcl}_P^X(S)$ and, for all $\alpha \in X$ and $p' \in \mathrm{bcl}_P^X(S)$, if $p \dashrightarrow^{\alpha} p'$, then $p \in \mathrm{bcl}_P^X(S)$. If X_P is one of I_P, O_P, Ω_P or A_P, then we also write $\mathrm{bcl}_P^X(S)$ instead of $\mathrm{bcl}_P^{X_P}(S)$.

Definition 3.40 (Illegal States)**.** The set of *illegal states* of an EMIA P is defined as $\mathrm{ill}_P := \mathrm{bcl}_P^{\Omega}(E_P \cup U_P) \setminus (E_P \cup U_P)$.

The set $\mathrm{ill}_{P \otimes Q}$ of an EMIA composition $P \otimes Q$ corresponds to the set of illegal states in IA [AH01a], MI [Rac+11] and MIA [Buj+16]. In contrast to these theories, EMIA requires one to match transitions of such states in the refinement $\sqsubseteq_{\mathrm{e}}$. The resulting refinement preorder is comparable to other refinement preorders for error-free interfaces, but is more detailed for erroneous ones. Indeed, MIA can be seen as an abstraction of EMIA, where all states in $\mathrm{ill}_{P \otimes Q}$ are deemed equivalent as we show in Theorem 3.48. For instance, the interfaces $P\colon \{i\}/\{o_1, o_2\}\colon p_0 \xrightarrow{i?} p_1 \xrightarrow{o_1!} \mathsf{err}_P$ and $Q\colon \{i\}/\{o_1, o_2\}\colon q_0 \xrightarrow{i?} q_1 \xrightarrow{o_2!} \mathsf{err}_Q$ are equivalent in MIA because after receiving input i, both may reach an error autonomously, whereas EMIA distinguishes P and Q according to the different behaviours ($o_1!$ vs. $o_2!$) that lead to an error.

Definition 3.41 (Error Abstraction)**.** The *error abstraction* of an EMIA $P \in \mathsf{EMIA}'$ is the EMIA $\alpha(P) := (S_{\alpha(P)}, I_P, O_P, \longrightarrow_{\alpha(P)}, \dashrightarrow_{\alpha(P)}, S^0_{\alpha(P)}, E_P, U_{\alpha(P)})$ with $S_{\alpha(P)} := (S_P \setminus \mathrm{ill}_P) \uplus \{\top_{\alpha(P)}\}$ and $U_{\alpha(P)} := U_P \uplus \{\top_{\alpha(P)}\}$. If $S^0_P \cap \mathrm{ill}_P \neq \emptyset$, then $S^0_{\alpha(P)} := (S^0_P \cap S_{\alpha(P)}) \cup \{\top_{\alpha(P)}\}$, else $S^0_{\alpha(P)} := S^0_P \cap S_{\alpha(P)}$. The transitions of $\alpha(P)$ are obtained from P as follows: if a state p specifies an $i?$-transition into ill_P, then all $i?$-transitions starting from p are replaced by a transition $p \xrightarrow{i?} \top_{\alpha(P)}$ and the underlying may-transition.

Obviously, $\alpha(P) \in \mathsf{MIA}$ for all $P \in \mathsf{EMIA}'$. Hence, by abuse of notation, we write α for both a function in $\mathsf{EMIA}' \to \mathsf{EMIA}'$ and in $\mathsf{EMIA}' \to \mathsf{MIA}$. Further, α is monotonic:

Lemma 3.42 (Monotonicity of α)**.** *The map α defined in Definition 3.41 is monotonic with respect to $\sqsubseteq_{\mathrm{e}}$.*

Proof. Let $\mathcal{R}$ be an error-preserving modal refinement relation between EMIAs P and Q. We show that the relation $\mathcal{R}_\alpha := (\mathcal{R} \cap (S_{\alpha P} \times S_{\alpha Q})) \cup (S_{\alpha P} \times U_{\alpha Q})$ is an error-preserving modal refinement relation between αP and αQ. Let $(p, q) \in \mathcal{R}_\alpha$. In case $q \in U_{\alpha Q}$, the definition of refinement is trivially satisfied, so we can assume $q \notin U_{\alpha Q}$. Hence, by definition of $\mathcal{R}_\alpha$, we may assume $(p, q) \in \mathcal{R}$ and distinguish the following cases:

R1, R2 Because $\mathcal{R}$ is an error-preserving modal refinement relation, $(p, q) \in \mathcal{R}$ trivially implies that R1 and R2 are satisfied.

R3 Let $q \xrightarrow{i}_{\alpha Q} Q'_\alpha$. We consider two cases:

1. The transition is due to a transition $q \dashrightarrow^{i}_{Q} q'$ with $q' \in \mathrm{ill}_Q$, i.e., $Q'_\alpha = \{\top_{\alpha Q}\}$: Any P'_α is a possible implementation of Q'_α.

2. The transition is due to a transition $q \xrightarrow{i}_Q Q'$: Because all transitions into ill_Q are replaced in Definition 3.41, we know that $Q'_\alpha = Q'$ and that none of these target states is in ill_Q or $E_Q \cup U_Q$. By $(p,q) \in \mathcal{R}$, there is a $p \xrightarrow{i}\!\!\!\!\rightarrow_P P'$ such that P' matches Q'. With the same argument as before, we may conclude that $P'_\alpha := P'$ matches Q'_α.

R4 Similar to R3(2), where $\xrightarrow{i}\!\!\!\!\rightarrow$ is replaced by $\rightarrowtail^{\omega}\!\!\!\!\rightarrow$.

R5 Let $p \dashrightarrow^{i}_{\alpha P} p'$. If $p' \neq \top_{\alpha P}$, then $p \dashrightarrow^{i}_P p'$ and, due to $(p,q) \in \mathcal{R}$, there is a $q \dashrightarrow\!\!\!\!\rightarrow^{i}_Q q'$ such that $(p',q') \in \mathcal{R}$. There are two cases:

1. $\exists q'' \in \mathrm{ill}_Q.\, q \dashrightarrow^{i}_Q q''$: By definition of α we have $\mathrm{ill}_Q \cap S_{\alpha Q} = \emptyset$; thus, $q'' \notin S_{\alpha Q}$. Hence, it follows from $q \in S_{\alpha Q}$ that $q \dashrightarrow^{i}_Q \top_{\alpha Q}$ by definition of α, and $(p', \top_{\alpha Q}) \in \mathcal{R}_\alpha$ is obvious.

2. $\forall q'' \in \mathrm{ill}_Q.\, q \not\dashrightarrow^{i}_Q q''$: The definition of α implies $q' \in S_{\alpha Q}$ and $q \dashrightarrow\!\!\!\!\rightarrow^{i}_{\alpha Q} q'$. Therefore, $(p',q') \in \mathcal{R}_\alpha$.

If $p' = \top_{\alpha P}$, then there is a $p'' \in \mathrm{ill}_P$ with $p \dashrightarrow^{i}_P p''$. By $(p,q) \in \mathcal{R}$, there exists a $q'' \in \mathrm{ill}_Q$ such that $q \dashrightarrow\!\!\!\!\rightarrow^{i}_Q q''$ and $(p'',q'') \in \mathcal{R}$. Thus, $q \dashrightarrow^{i}_{\alpha Q} \top_{\alpha Q}$, and $(\top_{\alpha P}, \top_{\alpha Q}) \in \mathcal{R}_\alpha$ is trivial.

R6 Analogous to R5 with $\dashrightarrow\!\!\!\!\rightarrow^{i}$ and $\dashrightarrow^{i}$ replaced by $\rightarrowtail^{\omega}\!\!\!\!\rightarrow$ and $\dashrightarrow^{\omega}$, respectively, and where we always have $p' \neq \top_{\alpha P}$ and only Case 2 applies (otherwise, we would have $q \in \mathrm{ill}_Q$). □

Now we can define MIA parallel composition and show that it is compositional.

Definition 3.43 (MIA Parallel Composition [Buj+16])**.** For composable MIAs P and Q, the *parallel product* is given by $P \otimes Q$ as defined in Definition 3.7. The *MIA-parallel composition* is defined as the MIA $P \parallel Q := \alpha(P \otimes Q)$.

Due to Lemma 3.38 and $\alpha(P)$ being a MIA, $P \parallel Q$ is also a MIA.

Lemma 3.44 (Compositionality of $\parallel$)**.** *If P_1, P_2 and Q are MIAs such that $P_1 \sqsubseteq_\mathrm{e} P_2$ and P_2, Q are composable, then P_1, Q are composable and $P_1 \otimes Q \sqsubseteq_\mathrm{e} P_2 \otimes Q$.*

Proof. Composability is obvious. By Theorem 3.9 and Lemma 3.42, we have $P_1 \parallel Q = \alpha(P_1 \otimes Q) \sqsubseteq_\mathrm{e} \alpha(P_2 \otimes Q) = P_2 \parallel Q$. □

Figure 3.6 illustrates why leading τs cannot be permitted in Rule R5 of Definition 3.5 when error-abstraction is used. If leading τs were permitted, then specification P would refine Q. However, $P \parallel R \not\sqsubseteq_\mathrm{m} Q \parallel R$ due to the universal states introduced by error-abstraction, i.e., the $\sqsubseteq_\mathrm{m}$ would not be compositional with respect to $\parallel$.

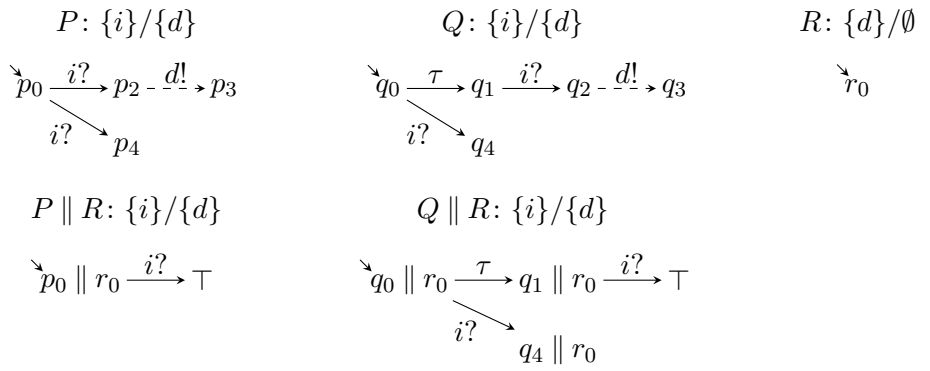


Figure 3.6.: Compositionality flaw with leading τ and error-abstraction.

Lemma 3.45 (α is Homomorphic with respect to Parallel Composition)**.** *The mapping α defined in Definition 3.41 is homomorphic with respect to parallel composition, i.e., $\alpha(P \otimes Q) \equiv_{\mathrm{m}} \alpha(P) \parallel \alpha(Q)$.*

Proof. First, observe that $\alpha(P \otimes Q)$ and $\alpha(P) \parallel \alpha(Q)$ have the same state set $S := S_{\alpha(P\otimes Q)} = S_{\alpha(P)\parallel\alpha(Q)}$, because the same pruning operation is used in α and in MIA's parallel composition operator (see also [Buj+16; BV14]).

"$\sqsubseteq_{\mathrm{m}}$": We show that the relation $\mathcal{R} := \mathrm{id}_S \cup (S_{\alpha(P\otimes Q)} \times \{\top_{\alpha(P)\parallel\alpha(Q)}\})$ is a MIA-refinement relation. Let $(s,t) \in \mathcal{R}$. If $t = \top_{\alpha(P)\parallel\alpha(Q)}$, there is nothing to show. Thus, we assume $s = t$ and distinguish the following cases:

R1, R2 From $s = t$ one directly concludes R1 and R2.

R3 Let $s = (p,q) \in S_{\alpha(P\parallel Q)}$. A transition $(p,q) \xrightarrow{i}_{\alpha(P)\parallel\alpha(Q)} S'$ is due to one of the rules P1, P2 or P3:

- **P1** $S' = P' \times \{q\}$ for some $p \xrightarrow{i}_{\alpha(P)} P'$ and $i \notin A_{\alpha(Q)}$: because this transition has neither been pruned nor replaced by a may-transition to $\top_{\alpha(P)\parallel\alpha(Q)}$, the same transition also exists in $\alpha(P \parallel Q)$.
- **P2** $S' = \{p\} \times Q'$ for some $q \xrightarrow{i}_{\alpha(Q)} Q'$ and $i \notin A_{\alpha(P)}$: Analogous to P1.
- **P3** $S' = P' \times Q'$ for some $p \xrightarrow{i}_{\alpha(P)} P'$ and $q \xrightarrow{i}_{\alpha(Q)} Q'$: Similar to P1 and P2.

R4 Analogous to R3.

R5 Let $(p,q) \dashrightarrow_{\alpha(P\parallel Q)} s''$. In case $s'' = \top_{\alpha(P)\parallel\alpha(Q)}$, then this transition is due to a replacement of a transition $(p,q) \dashrightarrow_{\alpha(P\parallel Q)} s'$ by $\top_{\alpha(P)\parallel\alpha(Q)}$. In case $s'' \neq \top_{\alpha(P)\parallel\alpha(Q)}$, by choosing $s' = s''$, we also have a transition $(p,q) \dashrightarrow_{\alpha(P\parallel Q)} s'$. In both cases, this transition is due to one of the rules P4 through P6, which all result in a similar line of argument. In the case of P4, we have $s' = (p',q)$, $p \dashrightarrow_{\alpha(P)} p'$ and $a \notin A_Q$. By the definition of α, there must be a p'' such that $p \dashrightarrow_P p''$. By P4, $(p,q) \dashrightarrow_{P\parallel Q} (p'',q)$ and, thus, also $(p,q) \dashrightarrow_{\alpha(P\parallel Q)} (p'',q)$.

R6 Analogous to R5.

Direction "$\sqsupseteq_{\mathrm{m}}$" can be shown dually. □

3.2.2. The Galois Insertion

The Galois insertion between MIA and EMIA consists of a concretisation $\gamma\colon \mathsf{MIA} \to \mathsf{EMIA}'$ and an abstraction $\alpha\colon \mathsf{EMIA}' \to \mathsf{MIA}$ such that (γ, α) is a Galois connection and $(\alpha \circ \gamma)(Q) \equiv_{\mathrm{m}} Q$. As presented in Section 3.2, the main idea behind α is to consider the states ill_P as equivalent. Each equivalence class of EMIAs resulting from this abstraction has a greatest element with respect to the refinement preorder, and α assigns each EMIA in such a class the greatest element of the class, which turns out to be a MIA. Vice versa, γ is the identical embedding of MIA into EMIA, such that a MIA represents its equivalence class.

Definition 3.46 (Concretisation Function from MIA to EMIA′)**.** The concretisation function $\gamma : \mathsf{MIA} \to \mathsf{EMIA}'$ is defined as $\gamma(P) := P$.

Obviously, γ is monotonic:

Lemma 3.47 (Monotonicity of γ)**.** *The map γ defined in Definition 3.46 is monotonic with respect to $\sqsubseteq_{\mathrm{m}}$ and $\sqsubseteq_{\mathrm{e}}$.*

The monotonicity of α and γ is key to the proof that α and γ form a Galois insertion:

Theorem 3.48 (Galois Insertion)**.** *The maps $\alpha\colon \mathsf{EMIA}' \to \mathsf{MIA}$ and $\gamma\colon \mathsf{MIA} \to \mathsf{EMIA}'$ defined in Definitions 3.41 and 3.46 form a Galois insertion between* MIA *and* EMIA′ *up to $\equiv_{\mathrm{m}}$, i.e., $P \sqsubseteq_{\mathrm{e}} \gamma(Q)$ iff $\alpha(P) \sqsubseteq_{\mathrm{m}} Q$ and $(\alpha \circ \gamma)(Q) \equiv_{\mathrm{m}} Q$ for all $P \in \mathsf{EMIA}'$ and $Q \in \mathsf{MIA}$.*

Proof. First, observe that $\alpha \circ \gamma = \mathrm{id}_{\mathsf{MIA}}$ (up to $\equiv_{\mathrm{m}}$). Second, the extensivity of α implies that $\gamma \circ \alpha$ is extensive, i.e., $P \sqsubseteq_{\mathrm{e}} (\gamma \circ \alpha)(P)$. Third, we show that α and γ form a Galois connection, i.e., $P \sqsubseteq_{\mathrm{e}} \gamma(Q)$ iff $\alpha(P) \sqsubseteq_{\mathrm{m}} Q$. Direction "⇒" holds due to $\alpha \circ \gamma = \mathrm{id}_{\mathsf{MIA}}$ and the monotonicity of α: $P \sqsubseteq_{\mathrm{e}} \gamma(Q) \Rightarrow \alpha(P) \sqsubseteq_{\mathrm{m}} (\alpha \circ \gamma)(Q) \equiv_{\mathrm{m}} Q$. Direction "⇐" follows from the monotonicity of γ, the extensivity of $\gamma \circ \alpha$ and the transitivity of $\sqsubseteq_{\mathrm{e}}$ by the following chain of implications: $\alpha(P) \sqsubseteq_{\mathrm{m}} Q \;\Rightarrow\; (\gamma \circ \alpha)(P) \sqsubseteq_{\mathrm{e}} \gamma(Q) \;\Rightarrow\; P \sqsubseteq_{\mathrm{e}} \gamma(Q)$. □

The extensivity of α makes γ non-homomorphic with respect to parallel composition; however, γ satisfies the inequality $\gamma(P \parallel Q) \sqsupseteq_{\mathrm{e}} \gamma(P) \otimes \gamma(Q)$ for MIAs P, Q. Although this follows directly from the definition of γ, we can prove a more general fact:

Lemma 3.49. *Let K and L be preorders, and $\cdot$ a binary operation on K respectively L. If (γ, α) is a Galois insertion between K and L such that α is homomorphic with respect to $\cdot$, then $\gamma(k \cdot k') \sqsupseteq \gamma(k) \cdot \gamma(k')$.*

Proof. $\gamma(k) \cdot \gamma(k') \sqsubseteq (\gamma \circ \alpha)(\gamma(k) \cdot \gamma(k')) \equiv \gamma((\alpha \circ \gamma)(k) \cdot (\alpha \circ \gamma)(k')) \equiv \gamma(k \cdot k')$. □

Corollary 3.50. *Let P and Q be MIAs. Then, $\gamma(P \parallel Q) \sqsupseteq_{\mathrm{e}} \gamma(P) \otimes \gamma(Q)$.*

Proof. By Theorem 3.48 and Lemma 3.45, we can apply Lemma 3.49. □

In the following lemmas 3.51 and 3.52 about abstraction and quotienting, we consider $\sqsubseteq_{\mathrm{e}}$ rather than $\sqsubseteq_{\mathrm{m}}$ because we do not know a priori that the quotient of MIAs is a MIA. In particular, the equivalence $\equiv_{\mathrm{e}}$ in Lemma 3.52 is not trivial because Lemma 3.51 only guarantees $\sqsubseteq_{\mathrm{e}}$ instead of $\equiv_{\mathrm{e}}$.

Lemma 3.51 (Abstraction Respects Quotienting)**.** *If $Q \sqsubseteq_{\mathrm{e}} P /\!\!/ D$ for EMIAs P, Q and D, then $\alpha(Q) \sqsubseteq_{\mathrm{e}} \alpha(P) /\!\!/ \alpha(D)$.*

Proof. We have $Q \sqsubseteq_{\mathrm{e}} P /\!\!/ D \overset{\text{Def. 3.27}}{\iff} Q \otimes D \sqsubseteq_{\mathrm{e}} P \overset{\text{Lem. 3.42}}{\implies} \alpha(Q \otimes D) \sqsubseteq_{\mathrm{e}} \alpha(P) \overset{\text{Lem. 3.45}}{\iff} \alpha(Q) \parallel \alpha(D) \sqsubseteq_{\mathrm{e}} \alpha(P) \overset{\text{Def. 3.27}}{\iff} \alpha(Q) \sqsubseteq_{\mathrm{e}} \alpha(P) /\!\!/ \alpha(D)$. □

Substituting $P /\!\!/ D$ for Q in Lemma 3.51 yields $\alpha(P /\!\!/ D) \sqsubseteq_{\mathrm{e}} \alpha(P) /\!\!/ \alpha(D)$.

Lemma 3.52 (MIA Quotient)**.** *Let P and D be MIAs. We have $P /\!\!/ D \equiv_{\mathrm{e}} \alpha(\gamma(P) /\!\!/ \gamma(D))$, i.e., MIA is closed under quotienting.*

Proof. By Lemma 3.51, Theorem 3.48, Definition 3.46 and extensivity of α (again Theorem 3.48), we get the following chain of inequalities: $\alpha(\gamma(P) /\!\!/ \gamma(D)) \sqsubseteq_{\mathrm{e}} \alpha(\gamma(P)) /\!\!/ \alpha(\gamma(D)) \equiv_{\mathrm{e}} P /\!\!/ D \equiv_{\mathrm{e}} \gamma(P) /\!\!/ \gamma(D) \sqsubseteq_{\mathrm{e}} \alpha(\gamma(P) /\!\!/ \gamma(D))$. By transitivity, all inequalities are equalities. □

3.2.3. Logical Operators under Galois Insertion

In this subsection we briefly discuss how the Galois insertion relates MIA and EMIA with respect to the logical operators presented in Section 3.1. First, we show that MIA is closed under conjunction.

Lemma 3.53 (MIA-Conjunction [Buj+16])**.** *If P and Q are MIAs with equal alphabets, then their conjunction $P \sqcap Q$ is also a MIA.*

Proof. It is easy to see that the conjunctive product & (see Definition 3.14) preserves the properties M1 through M3. Hence, it remains for us to show that the pruning of inconsistent states also preserves these properties. A state $p \sqcap q$ may only have a disabled input i if all i-may-transitions lead to states in $F_{\&}$. Then, $p \sqcap q$ would be inconsistent due to CF6, because $P \,\&\, Q$ satisfies M1 and M2. Therefore, $p \sqcap q$ must be input enabled. The same line of reasoning applies to M2. Property M3 is trivial. □

Hence, conjunction is the greatest lower bound with respect to $\sqsubseteq_{\mathrm{m}}$. The map α is not homomorphic with respect to conjunction: although $\alpha(P \sqcap Q) \sqsubseteq_{\mathrm{m}} \alpha(P) \sqcap \alpha(Q)$ holds for $P, Q \in \mathsf{EMIA}'$ because α is monotonic, the converse direction "$\sqsupseteq_{\mathrm{m}}$" does not hold in general, because MIA's replacement of illegal states by $\top$ must be reproduced by α. An example of EMIAs P and Q with $\alpha(P \sqcap Q) \not\sqsupseteq_{\mathrm{m}} \alpha(P) \sqcap \alpha(Q)$ is shown in Figure 3.7. State p_1 of specification P is in ill_P due to the b!-transition. Therefore, α prunes p_1 and replaces it by a universal state $\top$ in $\alpha(P)$. The conjunction $P \sqcap Q$ is inconsistent because P's regular state p_1 is conjoined with Q's fatal error state err, and the a?-must-transition

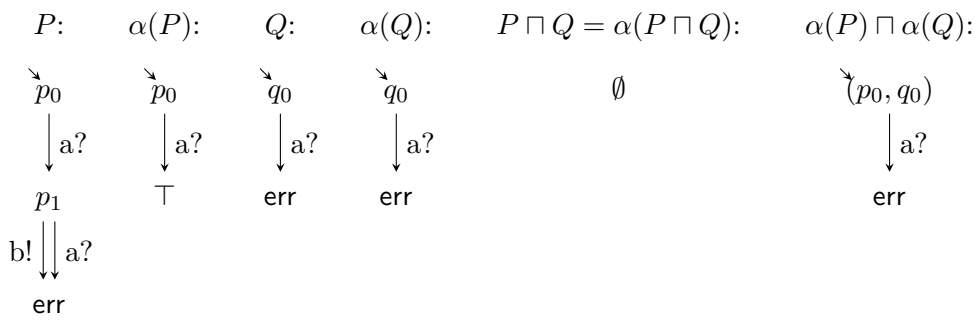


Figure 3.7.: Example of EMIAs P, Q with $\alpha(P \sqcap Q) \not\sqsupseteq_{\mathrm{m}} \alpha(P) \sqcap \alpha(Q)$, where $A = \{a\}/\{b\}$.

propagates this inconsistency back to the initial state. In the abstract setting, both the error and the inconsistency are avoided resulting in a regular and consistent initial state that is trivially refined by $P \sqcap Q$.

It is obvious that MIAs are closed under disjunction and that α is homomorphic with respect to disjunction. Further, α respects implication although we cannot define implication operationally:

Lemma 3.54 (Abstraction Respects Implication)**.** *Given EMIAs X, P and C, then $X \sqsubseteq_{\mathrm{e}} P \to C$ implies $\alpha(X) \sqsubseteq_{\mathrm{e}} \alpha(P) \to \alpha(C)$.*

Proof. $X \sqsubseteq_{\mathrm{e}} P \to C \iff X \sqcap P \sqsubseteq_{\mathrm{e}} C \iff X \sqsubseteq_{\mathrm{e}} C \land P \sqsubseteq_{\mathrm{e}} C \implies \alpha(X) \sqsubseteq_{\mathrm{e}} \alpha(C) \land \alpha(P) \sqsubseteq_{\mathrm{e}} \alpha(C) \iff \alpha(X) \sqcap \alpha(P) \sqsubseteq_{\mathrm{e}} \alpha(C) \iff \alpha(X) \sqsubseteq_{\mathrm{e}} \alpha(P) \to \alpha(C)$. □

3.3. Error-preserving vs. Error-abstracting Interface Theories

In this section we illustrate how the fatal error states employed in EMIA solve the issues I1–I4 of Section 2.3.3 present in error-abstracting interface theories such as IA [AH01a], MI [Rac+11] or MIA [Buj+16]:

I1. The possibility to redefine unwanted behaviour to not being an error when refining a composition impacts the usability of interface theories for safety-critical systems;
I2. The inability to check the compatibility of multiple components requires one to resort to separate assembly theories [HK15];
I3. A compatibility concept that is too strict for modelling product line families [LNW07];
I4. A global composition concept that requires a fixed view on composition.

Bujtor and Vogler [BV14] have shown that keeping or removing illegal states on a purely syntactic level are equivalent for IA with respect to preserving compatibility. In this spirit, current interface theories [Bau+10; Buj+16; BV14; AH01a; AH05; LNW07; LVF15; Rac+11] eliminate erroneous behaviour either by regarding it as undefined (pessimistic) or by pruning (optimistic); all errors are treated semantically equivalently. Due to this equivalence, theories combining IA and MTS cannot remove illegal states completely

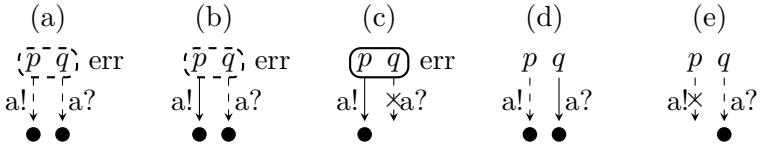


Figure 3.8.: Examples of errors: potential error (a) can be refined to potential error (b), to actual error (c) or resolved by refinement as in (d) or (e).

but must replace them by a special, arbitrarily refinable behaviour as is done in our error-abstraction function α of Definition 3.41.

However, because optional transitions (i.e., may-transitions) and disjunctive transitions allow for underspecification in MTS-based interface theories, one may distinguish potential errors that can be resolved by a suitable refinement from actual, unresolvable errors that arise when an output is required and the corresponding input is forbidden. For instance, specification (a) in Figure 3.8 shows a potential error: p might implement output a! whereas q does not guarantee that a? can be received in an implementation. This error can be refined to the potential error in (b) by turning the optional $a!$-transition in p into a mandatory $a!$-transition. It can be refined even further to the actual error in (c) by additionally forbidding the optional $a?$-transition in q. However, the potential error in (a) can also be resolved, e.g., by refining q's optional $a?$-transition into a mandatory $a?$-transition (d) or by removing p's optional $a!$-transition (e). An actual error as in (c) cannot be resolved by refining the components because it indicates that there is a concrete incompatibility between the components' specifications.

That is, specifications based on MTS contain more information with respect to compatibility, which we make explicit in EMIA. EMIA guarantees that compatible specifications have only compatible implementations, potential errors have both compatible and erroneous implementations, and actual errors have only erroneous implementations (see also Issue I3).

In particular, we establish in this section that EMIA treats unwanted behaviour more intuitively (Issue I1), that EMIA, in contrast to MIA, is an assembly theory (Issue I2), that EMIA provides better support for specifying product families (Issue I3), and that EMIA unifies the composition concepts of MTS and interface theories (Issue I4).

3.3.1. Issue I1: Unwanted Behaviour

In error-abstracting interface theories, forbidden inputs are preserved by the refinement preorder but are widely ignored by parallel composition, such that behaviour that is forbidden in one component may be re-introduced in the composed system if another component defies this prohibition. This unintuitive treatment of communication mismatches and, in particular, unwanted behaviour, is dangerous for safety-critical applications.

We illustrate this issue by reconsidering our running example of a driving assistance system and its communication with a garage in an IA-like interface theory such as IA, MI or MIA. Figure 3.9 shows specifications G and C of the garage's and the car's interfaces, respectively. Starting in state g_0, the garage is ready to receive a passage

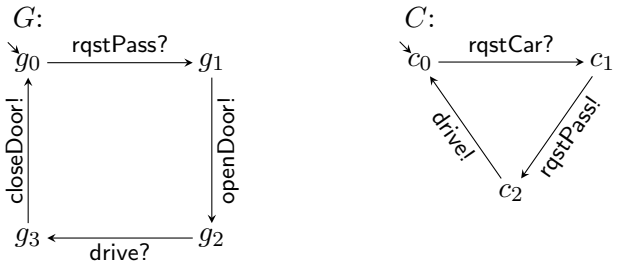


Figure 3.9.: The driving assistant system in IA or MIA including a garage G and a car C, where $A_G := \{\mathsf{drive?}, \mathsf{rqstPass?}\}/\{\mathsf{closeDoor!}, \mathsf{openDoor!}\}$ and $A_C := \{\mathsf{rqstCar?}\}/\{\mathsf{drive!}, \mathsf{rqstPass!}\}$.

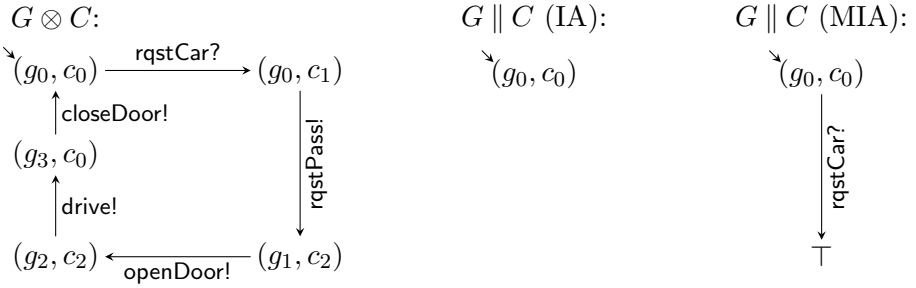


Figure 3.10.: Parallel product in IA or MIA (left), and parallel composition in IA (middle) and MIA (right) of the components depicted in Figure 3.9, where $A_{G\otimes C} = A_{G\parallel C} := \{\mathsf{rqstCar?}\}/\{\mathsf{drive!}, \mathsf{closeDoor!}, \mathsf{openDoor!}, \mathsf{rqstPass!}\}$.

request (rqstPass?). After such a request, the garage opens its door (openDoor!), waits for a car driving in or out (drive?) and, finally, closes the door (closeDoor!) again. The car starts in state c_0 waiting for a user's request (rqstCar?). Upon receiving such a request, the car requests passage from the garage (rqstPass!) and then drives into or out of the garage (drive!), reaching state c_0 again.

Specifications G and C have a communication mismatch due to the drive!-transition at state c_2 and the fact that no drive?-transition is specified at state g_1. Hence, in the parallel product $G \otimes C$ shown in Figure 3.10 (left), state (g_1, c_2) is considered illegal. In interface theories employing pessimistic compatibility, e.g., [Bau+10; LVF15], the parallel composition of G and C is undefined, because the illegal state (g_1, c_2) is reachable from the initial state (g_0, c_0). In contrast, optimistic theories, e.g., [AH01a; AH05; Buj+16; BV14; Che+12; LNW07; LVF15; Rac+11], consider a state optimistically illegal if a communication mismatch is reachable via uncontrollable actions, i.e., output or τ-transitions. The parallel composition $G \parallel C$ is obtained from $G \otimes C$ by removing all illegal states. In our example, state (g_1, c_2) is illegal, just as state (g_0, c_1) from which (g_1, c_2) is reachable by an output (rqstPass!). This pruning leaves a single state (g_0, c_0) with no transitions; all other states are unreachable. The rqstCar?-transition at state (g_0, c_0), which would allow one to reach illegal states when triggered by the environment, is also removed. However, in order to ensure compositionality of refinement,

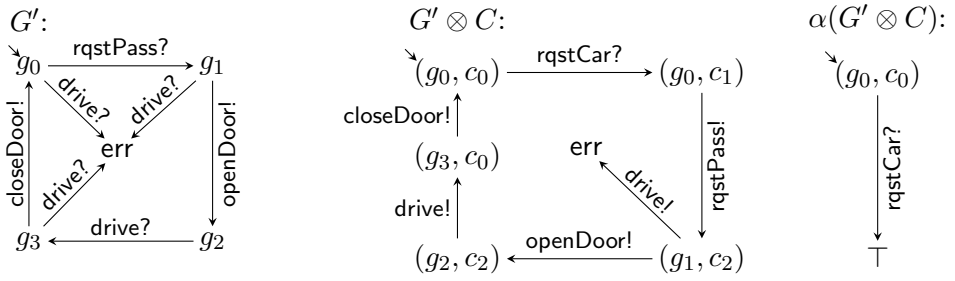


Figure 3.11.: Driving assistant system in EMIA and its Galois abstraction with the alphabets $A_{G'} := \{\mathsf{drive?}, \mathsf{rqstPass?}\}/\{\mathsf{closeDoor!}, \mathsf{openDoor!}\}$ and $A_{G' \otimes C} = A_{\alpha(G' \otimes C)} := \{\mathsf{rqstCar?}\}/\{\mathsf{closeDoor!}, \mathsf{drive!}, \mathsf{openDoor!}, \mathsf{rqstPass!}\}$.

rqstCar? must be permitted with arbitrary behaviour afterwards (see [Buj+16]); IA-based refinement [AH01a; AH05; LVF15] allows this implicitly for all unspecified inputs (Figure 3.10, middle). In MTS-based interface theories, where unspecified transitions represent forbidden behaviour, compositionality is achieved by replacing the pruned behaviour by an explicit optional transition to a special, universally refinable state $\top$ (Figure 3.10, right) that semantically stands for arbitrary behaviour [Buj+16]. This corresponds to the error-abstraction α of Section 3.2.

Due to this possibility of introducing arbitrary behaviour in case of a communication mismatch, stepwise refinement may re-introduce behaviour that has previously been removed due to the mismatch. Hence, optimistic theories accept a car driving into or out of the garage before the door is opened as a valid implementation of $G \parallel C$. This contradicts G's sensible requirement that driving in or out is only permitted after the door has been opened, i.e., the meaning of a car crashing into the door can simply be 'refined' to not being an error. In other words, the assumptions and guarantees expressible in current interface theories are insufficient for expressing unwanted behaviour.

In EMIA, the garage's constraint that a car shall not drive in or out in state g_1 would be specified by a drive?-transition to a fatal error state err, which represents an unresolvable error as is illustrated in specification G' in Figure 3.11. In the resulting parallel composition $G' \otimes C$, also shown in Figure 3.11, driving in or out too early in state (g_1, c_2), when the door is still closed, leads to the fatal error state err, where the car crashes into the door. This information is not removed and cannot be redefined to not being an accident by refining $G' \otimes C$. Keeping this information is essential for pinning down the location and the cause of the error within the specification. Because G' forbids action drive? between rqstPass? and openDoor! but allows drive? after openDoor!, we can infer that specification C must be aware of action openDoor! in order to be compatible with G'. This way, a software design tool based on EMIA could propose possible specification changes to the designer. For example, the tool may propose to add action openDoor? to the car's alphabet and to insert an openDoor?-transition between rqstPass! and drive!, so as to avoid the fatal error state err that is reachable from (g_1, c_2). The resulting specification is shown as C' in Figure 3.12.

However, when applying the abstraction function α of the Galois insertion between

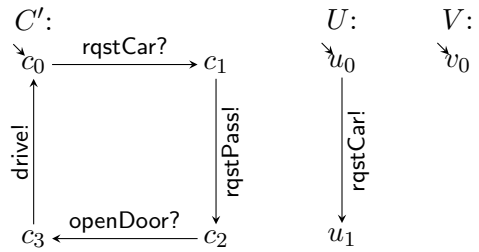


Figure 3.12.: Corrected car C' and user interfaces U, V, where $A_{C'} := \{\mathsf{openDoor?}, \mathsf{rqstCar?}\}/\{\mathsf{drive!}, \mathsf{rqstPass!}\}$ and $A_U = A_V := \emptyset/\{\mathsf{rqstCar!}\}$.

MIA and EMIA in order to move into the error-abstracting setting, all the information about errors is lost. Figure 3.11 (right) illustrates the Galois abstraction of the composed system $G' \parallel C$. We have $\mathrm{ill}_{G' \parallel C} = \{(g_1, c_2), (g_0, c_1)\}$ (see Section 3.2). The rqstCar?-must-transition at (g_0, c_0) leading to $\mathrm{ill}_{G' \parallel C}$ is replaced by a rqstCar?-transition to $\top_{\alpha(G' \parallel C)}$. Due to α being a homomorphism with respect to $\parallel$, this result corresponds exactly to the MIA shown in Figure 3.10 (right).

3.3.2. Issue I2: Multi-component Assemblies

Pairwise binary compatibility of multiple components is neither necessary nor sufficient for their overall compatibility when being considered as a multi-component assembly, even if parallel composition is associative. To address this, Hennicker and Knapp [HK15] have introduced *assembly theories* that extend interface theories by a separate level of assemblies where multi-component compatibility is checked. However, these assemblies have to be re-interpreted as interfaces to be of further use.

When adding the specification of a simple user interface, shown as U in Figure 3.12, as a third component to the specifications G and C of Figure 3.9, the three components G, C and U are pairwise optimistically compatible. However, the composed system $G \parallel C \parallel U$ is incompatible, because the mismatch for action drive! is reachable from the initial state $\langle g_0, c_0, u_0 \rangle$. A different but related problem arises in pessimistic theories due to weak associativity: the user interface specification V in Figure 3.12 promises to never request a car. The components G and C are pessimistically incompatible and $(G \parallel C) \parallel V$ is undefined. However, $G \parallel (C \parallel V)$ is a perfectly valid composition. In other words, pairwise compatibility is neither necessary nor sufficient for compatibility of multiple components, i.e., IA, MI, MIO and MIA are not by themselves assembly theories.

To lift their interface theory MIO to an assembly theory, Hennicker and Knapp propose an enrichment EMIO of MIO by error states similar to our fatal errors [HK15]. However, they do not develop EMIO into a full interface theory: EMIOs are only employed to describe the result of a multi-component parallel composition and to check the communication safety of such an assembly, i.e., the absence of communication mismatches. In addition, refinement is lifted to assemblies by providing an error-preserving refinement relation for EMIOs, which is similar to error-preserving modal refinement. However, no further operations like parallel composition or conjunction are defined for assemblies;

instead, EMIO forms a second layer on top of MIO, and an EMIO is re-interpreted as MIO via an encapsulation function that removes all error-information. In contrast to this loose integration, EMIA provides a uniform and tight integration of interfaces and assemblies by directly including its canonical assembly theory in the sense of [HK15]. In particular, EMIA does not need two separate refinement relations for interfaces and assemblies.

Translating the above examples of assemblies with U and V into EMIA, the composition $G' \otimes C \otimes U$ resembles $G' \otimes C$ (Figure 3.11), except that action rqstCar is an output instead of an input. Further, $(G' \otimes C) \otimes V$ and $G' \otimes (C \otimes V)$ are equivalent in EMIA. In both examples, compatibility is checked via reachability of fatal error states in the composed system. However, it is up to the systems designer to decide which error behaviour yields an incompatibility, i.e., compatibility is not necessarily a globally predefined concept as is the case for optimistic and pessimistic compatibility.

In order to establish EMIA as an assembly theory, we recap the definition of *assembly theory* by Hennicker and Knapp [HK15], with the following generalisation: in Hennicker and Knapp's definition of an interface theory, an interface cannot contain errors by itself and, thus, a single interface is always communication safe. EMIA additionally allows one to specify erroneous interfaces, which should not be considered communication safe. Therefore, we introduce a *communication safety predicate* cs on interfaces and generalise Conditions A1 and A3 below accordingly. This predicate also generalises the binary *compatibility predicate* cp employed in [HK15] by defining $\mathrm{cp}(P, Q) :\iff \mathrm{cs}(P \otimes Q)$. We use the same symbol cs for interfaces and assemblies because it will always be clear from the context whether the interface-theoretic or the assembly-theoretic communication safety is meant.

Definition 3.55 (Assembly Theory [HK15])**.** Let $\mathfrak{I} := (\mathcal{I}, \mathrm{cs}, \|, \sqsubseteq)$ be an interface theory, where $\mathcal{I}$ is a class of interfaces, $\mathrm{cs} \subseteq \mathcal{I}$ is a communication safety predicate, $\|$ is a (binary) parallel composition operator, and $\sqsubseteq$ is the refinement preorder. A tuple $\mathfrak{A} := (\mathcal{A}, \mathrm{cs}, \varphi, \preceq)$ consisting of a collection of *assemblies* $\mathcal{A} := \{\langle I_k \rangle_{k \in K} \mid 0 < |K| < \infty$ and $I_k, I_l \in \mathcal{I}$ composable for $k \neq l\}$, a *communication safety predicate* $\mathrm{cs} \subseteq \mathcal{A}$, a partial *encapsulation operation* $\varphi : \mathfrak{A} \rightharpoonup \mathfrak{I}$ and an *assembly refinement relation* $\preceq \subseteq \mathcal{A} \times \mathcal{A}$ is called an *assembly theory over* $\mathfrak{I}$ if, for all $A, B, A_1, \ldots, A_n, B_1, \ldots, B_n \in \mathcal{A}$ (where $n \in \mathbf{N}$) and $I, J \in \mathcal{I}$, we have:

A1. $\mathrm{cs}(\langle I \rangle)$ iff $\mathrm{cs}(I)$,
A2. if $\mathrm{cs}(A)$, then $\varphi(A)$ is defined,
A3. if $\varphi(\langle I \rangle)$ is defined, then $\varphi(\langle I \rangle) = I$,
A4. $\preceq$ is reflexive and transitive,
A5. $I \sqsubseteq J$ implies $\langle I \rangle \preceq \langle J \rangle$,
A6. if $A = A_1 \uplus \cdots \uplus A_n$ and $\mathrm{cs}(A_k)$ for $k = 1, \ldots, n$, then $\langle \varphi(A_1), \ldots, \varphi(A_n) \rangle \in \mathcal{A}$,
A7. if $A = A_1 \uplus \cdots \uplus A_n$, $\mathrm{cs}(A_k)$ for $k = 1, \ldots, n$ and $\mathrm{cs}(\langle \varphi(A_1), \ldots, \varphi(A_n) \rangle)$, then $\mathrm{cs}(A)$,
A8. if $A = A_1 \uplus \cdots \uplus A_n$, $\mathrm{cs}(A_k)$ for $k = 1, \ldots, n$ and $\mathrm{cs}(\langle \varphi(A_1), \ldots, \varphi(A_n) \rangle)$, then $\varphi(A) = \varphi(\langle \varphi(A_1), \ldots, \varphi(A_n) \rangle)$,

A9. if $A \preceq B$ and $\mathrm{cs}(B)$, then $\mathrm{cs}(A)$,
A10. if $A \preceq B$ and $\mathrm{cs}(B)$, then $\varphi(A) \sqsubseteq \varphi(B)$,
A11. if $A = A_1 \uplus \cdots \uplus A_n$, $B = B_1 \uplus \cdots \uplus B_n$, $\mathrm{cs}(\langle \varphi(B_1), \ldots \varphi(B_n) \rangle)$, as well as $\mathrm{cs}(B_k)$ and $A_k \preceq B_k$ for $k = 1, \ldots, n$, then $A \preceq B$.

Rule A1 expresses that a single-component assembly is communication safe if and only if the underlying component is communication safe. Rule A2 ensures that any communication safe assembly may be encapsulated into an interface. By Rule A3, the interface of a single-component assembly is equivalent to this single component. Rule A4 states that assembly-refinement shall be a preorder. Rule A5 expresses that interface refinement translates to refinement of single-component assemblies. Rule A6 means that assemblies may be constructed hierarchically. Rule A7 expresses that communication safety is compositional. Rule A8 says that assemblies may be encapsulated according to the hierarchy of their construction. Rule A9 requires that assembly-refinement preserves communication safety. Rule A10 says that encapsulation is monotonic for communication safe assemblies. Rule A11 ensures the compositionality of assembly-refinement with respect to the construction of assemblies, i.e., the substitutability of components by refinements thereof.

Intuitively, the encapsulation $\varphi(A)$ of an assembly A represents the composition of A's components as an interface. Therefore, an assembly theory is called *canonical* if there is a strong correspondence between φ and $\|$. We write $\prod_{k \in K}$ for the generalisation of $\|$ to multiple interfaces.

Definition 3.56 (Canonical Assembly Theory [HK15])**.** An assembly theory is called *canonical* if the following conditions hold:

1. $\mathrm{cs}(\langle I_k \rangle_{k \in K})$ iff, for all $l \in K$, $\mathrm{cs}(I_l \parallel \prod_{k \in K \setminus \{l\}} I_k)$,
2. $\varphi(\langle I_k \rangle_{k \in K}) = \prod_{k \in K} I_k$ if $\mathrm{cs}(\langle I_k \rangle_{k \in K})$, and undefined otherwise.

It is straightforward to define a canonical assembly theory over EMIA:

Definition 3.57 (Assembly Theory over EMIA)**.** Let $\mathfrak{I}_{\mathsf{EMIA}} := (\mathsf{EMIA}, \mathrm{cs}, \otimes, \sqsubseteq_{\mathrm{e}})$ with $\mathrm{cs}(I)$ iff $S^0_I \cap \mathrm{bcl}^{\Omega}_I(E_I \cup U_I) = \emptyset$. We define $\mathfrak{A}_{\mathsf{EMIA}} := (\mathcal{A}, \mathrm{cs}, \varphi, \preceq)$ by:

1. $\mathcal{A} := \{\langle I_k \rangle_{k \in K} \mid 0 < |K| < \infty$ and $I_k, I_l \in \mathsf{EMIA}$ composable for $k \neq l\}$,
2. $\mathrm{cs}(A)$ iff $S^0_{\varphi(A)} \cap \mathrm{bcl}^{\Omega}_{\varphi(A)}(E_{\varphi(A)} \cup U_{\varphi(A)}) = \emptyset$,
3. $\varphi(\langle I \rangle) := I$ and $\varphi(\langle I_1, \ldots, I_n \rangle) := I_1 \otimes \ldots \otimes I_n$, and
4. $A \preceq B$ iff $\varphi(A) \sqsubseteq_{\mathrm{e}} \varphi(B)$.

Note that the interface-level predicate cs employed in Definition 3.57 corresponds to optimistic compatibility. Due to the flexibility of EMIA, a pessimistic variant may easily be defined by $\mathrm{cs}(I) \iff S^0_I \cap \mathrm{bcl}^{A \cup \{\tau\}}_I(\mathrm{ill}_I) = \emptyset$ (cf. Definition 3.40).

Lemma 3.58. $\mathfrak{A}_{\mathsf{EMIA}}$ *is an assembly theory over* $\mathfrak{I}_{\mathsf{EMIA}}$*.*

Proof. A1 holds by definition. A2 is trivial because φ is defined for all assemblies. A3 holds by definition. A4 is trivial because $\sqsubseteq_{\mathrm{e}}$ is reflexive and transitive. A5 holds by definition. A6, A7 and A8 are trivial due to the associativity of EMIA parallel composition and the definition of the encapsulation function φ. A9 holds by definition of $\sqsubseteq_{\mathrm{e}}$. A10 holds by definition of $\preceq$. A11 holds due to the compositionality of $\sqsubseteq_{\mathrm{e}}$. □

$\mathfrak{A}_{\mathsf{EMIA}}$ obviously satisfies the first condition of Definition 3.56. It almost satisfies the second condition, except that instead of being undefined in the 'otherwise'-branch, an erroneous interface results from the composition. We can either artificially set such a result to undefined in order to match the definition exactly, or argue that undefinedness is only necessary here because interface theories in [HK15] do not support the specification of erroneous interfaces (and, thus, one may change that definition accordingly). In both cases we have:

Theorem 3.59 (Assembly Theory)**.** $\mathfrak{A}_{\mathsf{EMIA}}$ *is a canonical assembly theory over* $\mathfrak{I}_{\mathsf{EMIA}}$*.*

Because the encapsulation function φ directly corresponds to $\otimes$ and $\preceq$ corresponds to $\sqsubseteq_{\mathrm{e}}$, $\mathfrak{I}_{\mathsf{EMIA}}$ includes its own assembly theory $\mathfrak{A}_{\mathsf{EMIA}}$. In addition, EMIA obviously constitutes an assembly theory over MIA:

Definition 3.60 (Assembly Theory over MIA)**.** Let $\mathfrak{I}_{\mathsf{MIA}} := (\mathsf{MIA}, \mathrm{cs}, \|, \sqsubseteq_{\mathrm{m}})$ with $\mathrm{cs}(I)$ for all $I \in \mathsf{MIA}$. We define $\mathfrak{A}_{\mathsf{MIA}} := (\mathcal{A}, \mathrm{cs}, \varphi, \preceq)$ by:

1. $\mathcal{A} := \{\langle I_k \rangle_{k \in K} \mid 0 < |K| < \infty$ and $I_k, I_l \in \mathsf{MIA}$ composable for $k \neq l\}$,
2. $\mathrm{cs}(\langle I_k \rangle_{k \in K})$ iff $S^0_{I_1 \otimes \ldots \otimes I_K} \cap \mathrm{bcl}^{\Omega}_{I_1 \otimes \ldots \otimes I_K}(E_{I_1 \otimes \ldots \otimes I_K} \cup U_{I_1 \otimes \ldots \otimes I_K}) = \emptyset$,
3. $\varphi(\langle I \rangle) := I$ and $\varphi(\langle I_1, \ldots, I_n \rangle) := I_1 \parallel \ldots \parallel I_n$, and
4. $A \preceq B$ iff $\varphi(A) \sqsubseteq_{\mathrm{m}} \varphi(B)$.

Lemma 3.61. $\mathfrak{A}_{\mathsf{MIA}}$ *is a canonical assembly theory over* $\mathfrak{I}_{\mathsf{MIA}}$*.*

Proof. Obvious. □

3.3.3. Issue I3: Software Product Lines

Optional behaviour, modelled via may-transitions as in MTS, may be employed to express variability inherent in software product lines. In current interface theories, two product families may be considered compatible only if all products of one family are compatible with all products of the other. However, one would prefer a more detailed set of guarantees, such that one may distinguish if all, some or none of the product lines' products are compatible [LNW07] in order to compute compatible subfamilies.

Consider specifications D and W of a car and a user interface product family, respectively, both of which are shown in Figure 3.13. These specifications allow product variations of a car and a user interface, respectively, which enable drivers to initiate the automatic driving assistance manually (go!), e.g., when parking in a different garage

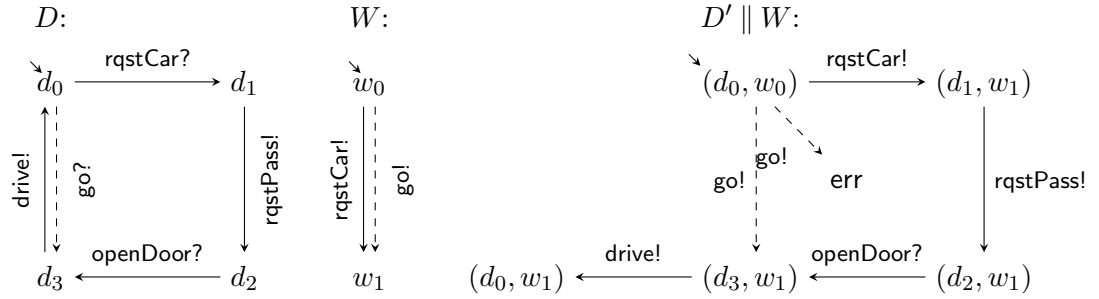


Figure 3.13.: Product families D and W and their composition in EMIA, where $A_D := \{\mathsf{go?}, \mathsf{openDoor?}, \mathsf{rqstCar?}\}/\{\mathsf{drive!}, \mathsf{rqstPass!}\}$, $A_W := \emptyset/\{\mathsf{go!}, \mathsf{rqstCar!}\}$ and $A_{D' \| W} := \{\mathsf{openDoor?}\}/\{\mathsf{drive!}, \mathsf{go!}, \mathsf{rqstCar!}, \mathsf{rqstPass!}\}$.

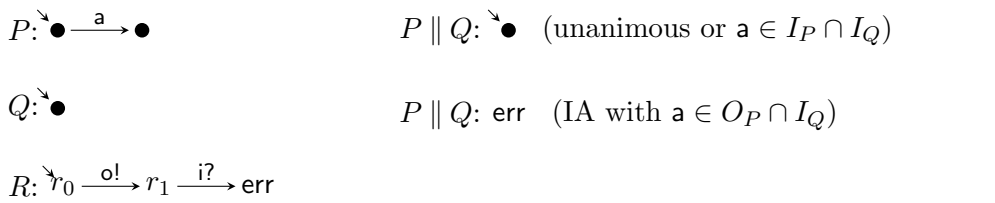


Figure 3.14.: Unanimous and error-aware composition of components with alphabets $A_P = A_Q = \{\mathsf{a}\}$ and $A_R = \{\mathsf{i}\}/\{\mathsf{o}\}$.

that is not equipped with an automatic door opener. Obviously, a user interface that provides this feature is incompatible with a car that does not, i.e., although some product combinations of D and W are compatible, some of them are not. Hence, D and W are incompatible, and no information that might help finding compatible product combinations is provided in current interface theories (see also the discussion about actual and potential errors in the introduction of this section).

In EMIA, the optional go?-transition at state d_0 would be modelled as a disjunctive go?-must-transition from d_0 to $\{d_3, \mathsf{err}\}$, for a fatal error state err. We refer to this specification as D'. The specified error information is still present in the parallel composition of D' and W, so that one may derive additional conditions on the go-transitions. These conditions result in compatible refinements of D' and W, which describe compatible sub-families of the original product families. For example, refining the optional go?-transition into a mandatory one in D', or removing the optional go!-transition in W; both result in appropriate restrictions to sub-families. The necessary error information is present in the EMIA parallel composition of D' and W (Figure 3.13).

3.3.4. Issue I4: Unified Composition Concepts

MTS and interface theories combining IA with MTS share many aspects of the modality semantics with respect to refinement. However, the meaning of may- and must-modalities differs with respect to parallel composition. Required and forbidden actions never cause

an error in a parallel composition in MTS: either all components *unanimously* agree on implementing an action, or the action is forbidden in the composed system. For example, when composing specifications P and Q from Figure 3.14 (left), action a is forbidden in the composition $P \parallel Q$ (Figure 3.14, top right). The possibility to disagree on transitions enables an environment to control all transitions of an MTS, such that they may be interpreted as input transitions from an interface-theoretic view, i.e., if $\mathsf{a} \in I_P \cap I_Q$. However, the MTS parallel composition is not directly applicable to output actions, because these cannot be controlled by the environment. Consequently, previous interface theories have adopted an IA-like *error-aware* parallel composition that considers $P \parallel Q$ erroneous if $\mathsf{a} \in O_P \cap I_Q$ (Figure 3.14, middle right) and is tightly bound to a fixed compatibility concept, such as optimistic or pessimistic compatibility. The choice of composition and compatibility concept is *global* and does not allow one to mix different such concepts according to what is suitable for the application at hand. In contrast, one of our goals for EMIA is to provide a general-purpose semantic theory that is independent of such fixed compatibility and composition concepts. EMIA's explicit error representation allows for a *local* description of compatibility that is independent of composition. For example, a composition with specification R (Figure 3.14, left) is unanimous with respect to input i in state r_0 and error-aware in state r_1. Thus, EMIA unifies unanimous and error-aware parallel composition, i.e., it permits the mixing of these composition concepts within a specification. As an aside, note that EMIA collapses to MTS when considering input actions only.

In particular, the traditional notions of optimistic and pessimistic compatibility may still be expressed in EMIA:

Definition 3.62 (Optimistic and Pessimistic Compatibility)**.** Two composable EMIAs P, Q are *optimistically compatible* if and only if $S^0_{P\otimes Q} \cap \mathrm{bcl}^{\Omega}_{P\otimes Q}(E_{P\otimes Q} \cup U_{P\otimes Q}) = \emptyset$. Further, P and Q are *pessimistically compatible* if and only if $S^0_{P\otimes Q} \cap \mathrm{bcl}^{A\cup\{\tau\}}_{P\otimes Q}(\mathrm{ill}_{P\otimes Q}) = \emptyset$.

As explained in Issue I1, the error-information is not removed, i.e., in an optimistic variant of EMIA one cannot introduce unwanted behaviour as is the case in previous optimistic theories.

3.4. Discussion

Of course, there are alternative possibilities for defining some of the details of our theory. In this section we discuss some of these alternatives and how they influence the theory. In particular, we investigate whether the alternatives break some of the desired properties. In addition, we discuss the complexity of operators on EMIAs and give a brief outlook on the integration of temporal logic operators into EMIA.

3.4.1. Refinement and Alphabet Extension

In the original work on IA [AH01a; AH05], a different notion of refinement is employed, which has also been investigated in the context of MIA in [LVF15]. Recall that an

incompatibility arises when one component specifies an output whereas another component does not require the corresponding input. The IA refinement is based on the intuition that removing outputs or adding missing inputs may only reduce the risk of having incompatibilities. In particular, an unspecified input represents an input that allows for arbitrary subsequent behaviour. The formalisation presented in the literature is obtained from Definition 3.5 by omitting Rule R5, which is equivalent to replacing Rule R5 by R5':

R5. $p \dashrightarrow^{i}_{P} p'$ implies $\exists q'.\, q \dashrightarrow^{i}_{Q} q' \wedge (p', q') \in R$,

R5'. $p \dashrightarrow^{i}_{P} p'$ implies $\top$.

As a consequence, may-transitions are irrelevant for input actions and one cannot specify optional input transitions. This refinement preorder is compositional if one requires that interfaces are input deterministic.

This determinism requirement is due to a misconception when formalising the intuition that unspecified inputs represent inputs with arbitrary subsequent behaviour. The mistake is that adding inputs is always permitted, no matter whether the same input is already specified. However, the above intuition only applies for *unspecified* inputs. If an action is already present, additional transitions of this action may of course introduce new incompatibilities. In order to formalise the alternative refinement correctly, we have to replace Rule R5 by

R5". $p \dashrightarrow^{i} p'$ implies $q \not\dashrightarrow^{i}$ or $\exists q'.\, q \dashrightarrow^{i} q' \wedge (p', q') \in R$.

With this corrected definition, an unspecified input is equivalent to an input may-transition targeting a universal state, and input determinism is not required for compositionality.

A different problem with this alternative definition of refinement is that conjunction leads to inadequate artefacts when the alphabets of the conjuncts must be extended [LVF15]. The main source of problems is due to the inability to specify optional inputs when employing this notion of refinement. For example, if conjunct P specifies a transition $p \xrightarrow{i} P'$ where input i is unknown to Q then $p \sqcap q \xrightarrow{i} P'$ for any $q \in Q$ because input i is required at p and implicitly permitted in q. Thereby, one loses the behavioural requirements expressed by conjunct Q. This is usually undesired in perspective-based specification. To keep conjunct Q, one could add an i-must-loop to each state $q \in Q$ such that then $p \sqcap q \xrightarrow{i} \{p' \sqcap q \mid p' \in P'\}$, expressing the neutrality of Q with respect to i in each of its states. But now the problem is at those states $p'' \in P$ with $p'' \not\dashrightarrow^{i}$, i.e., p'' stipulates on the environment not to produce i, because $p'' \sqcap q \xrightarrow{i} q$ is also undesirable. However, this problem may be resolved in EMIA where optional transitions may be expressed with the help of fatal error states, i.e., by adding a transition $q \xrightarrow{i} \{q, \mathsf{err}\}$ to each state q (where originally $q \not\dashrightarrow^{i}$) as in Definition 3.35.

A different approach to alphabet extension is presented in [BV16] for MIA. There, the action alphabets I and O are enriched by two special actions ν_I and ν_O that represent unknown input and output actions, respectively. With ν-may-transitions, an interface specifies the permitted behaviour with respect to new actions, e.g., a transition $p \dashrightarrow^{\nu_I} p'$ permits one to refine state p by adding arbitrary new input actions with subsequent

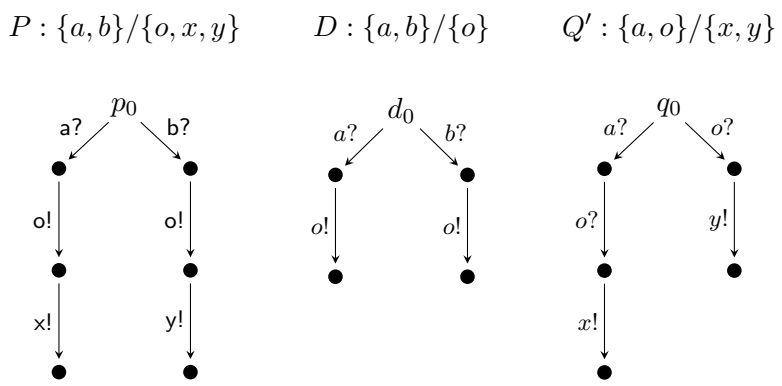


Figure 3.15.: Complications of quotienting in the context of alphabet extension.

behaviour p'. Such a specification is local to each state but global for all unknown actions. In particular, two components of a system may not be extended by the same new actions. Otherwise, compositionality of the refinement preorder with respect to parallel composition is not guaranteed [BV16].

3.4.2. Quotient

In Section 3.1.7 we restrict the choice of alphabet of the quotient and require that divisors are deterministic. In this section we explain the difficulties that arise when trying to relax these restrictions [Buj+16] and discuss related work.

Releasing Alphabet Restriction

For $Q \parallel D \sqsubseteq_{\mathrm{e}} P$ to hold, $Q \parallel D$ and P must have the same input alphabet and the same output alphabet; also such Q, D must be compatible. Thus, we have $O_Q = O_P \setminus O_D$ and $I_Q \supseteq I_P \setminus I_D$. Concerning the actions of D, quotient Q may listen to them but does not have to. Hence, $I_Q \subseteq (I_P \setminus I_D) \cup A_D = I_P \cup O_D$. The more inputs Q has, the easier it is to supply the behaviour ensuring $Q \parallel D \sqsubseteq_{\mathrm{e}} P$. Thus, we have chosen the largest possible input alphabet $I_P \cup O_D$ for our quotient $P /\!\!/ D$, just as in [CJK13] and [Rac+11].

When extending refinement via alphabet extension one may compare EMIAs with different alphabets. Hence, one could aim for a generalisation of Theorem 3.27 where Q and $P /\!\!/ D$ may have different alphabets. Because $Q \sqsubseteq_{\mathrm{e}} P /\!\!/ D$, the quotient should have a minimal alphabet in this version, in contrast to our choice of $I_{P/\!\!/D} = I_P \cup O_D$. However, this leads to complications as one can see from the example in Figure 3.15. An EMIA Q satisfying $Q \parallel D \sqsubseteq_{\mathrm{e}} P$ must have $O_Q = \{x, y\}$, but $I_Q = I_P \setminus I_D = \emptyset$ clearly does not suffice because Q is allowed to produce x or y only after o. Furthermore, Q must see a or b to distinguish between the branches. Solutions are possible, e.g., for $I_Q = \{a, o\}$ and $I_Q = \{b, o\}$; a solution Q' for $\{a, o\}$ is also shown in Figure 3.15, where transitions to the universal state are not drawn for readability. It looks like there are several maximal solutions.

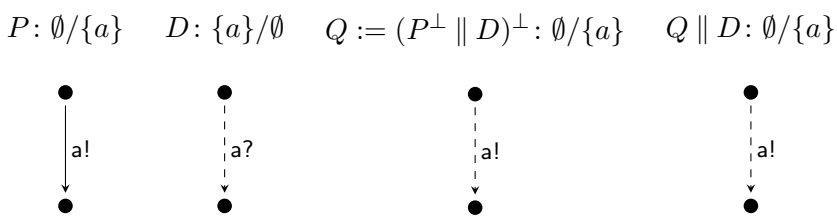


Figure 3.16.: The dual construction of the quotient is incorrect for modal transitions.

Note, however, that Theorem 3.27 in its present form still holds for our extended refinement preorder. This is important in practice where one would want for Q and D to be able to communicate via new internal actions, i.e., those that are hidden immediately after taking the parallel composition of Q and D. Since only outputs can be hidden, the new actions must form a set O' of outputs in $Q \parallel D$. Then, one proceeds by determining $Q := [P]_{\emptyset,O'} /\!/ D$. Theorem 3.27 implies $Q \parallel D \sqsubseteq_\mathrm{e} [P]_{\emptyset,O'}$, which in turn implies $(Q \parallel D) / O' \sqsubseteq_\mathrm{e} P$ by Proposition 3.13(1).

Another aspect of alphabet extension for quotienting is that we can generalise the problem by permitting D to have actions unknown to P. A straightforward generalisation of our approach in Section 3.1.7 would make these actions inputs for the quotient, but there can also be solutions to $Q \parallel D \sqsubseteq_\mathrm{e} P$ where Q has some new inputs of D as outputs. We leave a further investigation of these aspects to future work.

Related Work

Quotient operators for interface theories are also discussed by Raclet [Rac08], Bhaduri and Ramesh [BR08], Raclet et al. [Rac+11] and Chilton [Chi13].

Bhaduri and Ramesh [BR08] present an elegant construction of quotients for deterministic interface automata. If $P^\perp$ denotes the interface automaton P with alphabets I and O interchanged, then the quotient may be constructed as $P /\!/ D \equiv (P^\perp \parallel D)^\perp$. However, their work does not consider internal transitions and modalities. Figure 3.16 shows that this dual construction is incorrect when modal transitions are involved: the parallel composition $Q \parallel D$ of the quotient $Q := (P^\perp \parallel D)^\perp$ and the divisor D does not refine P.

Our quotient $Q = P /\!/ D$ is most similar to the one in MI [Rac+11], where D is assumed to be may-deterministic, P and D have no internal transitions, and $I_Q = I_P \cup O_D$. However, P must also be may-deterministic in [Rac+11], whereas we also allow nondeterminism and disjunctive must-transitions in P. In addition, we have corrected some technical shortcomings of MI. MI adapts the quotient operation for Modal Specifications from [Rac08], with some additional rules defining the input and output alphabets of the quotient interface. However, compatibility is ignored for the quotient operation, which in [Rac+11] is an inverse or adjoint to their parallel product but *not* to parallel composition. This has been recognised in a technical report [Ben+12]. Unfortunately, that report employs a changed setting without the universal state as in [Rac+11] or a universal state as in our work. This is reflected by a different, non-compositional parallel composition that does not allow arbitrary behaviour in case of an

inconsistency and that employs a more aggressive pruning strategy, where a mismatch can also occur when synchronising on shared inputs, i.e., when one component specifies input i and another component does not require input i.

Beneš et al. [Ben+13] investigate quotienting for nondeterministic specifications in the setting of DMTS. This comes with an exponential blowup in the quotient size, because the powerset of quotient states is used as state space. In principle, a similar solution would be necessary in order to relax our determinism requirement on the divisor. However, it is not straightforward to adopt this solution in the context of internal transitions and input/output with the related compatibility issues, which are core ingredients of interface theories being present since the first publications on IA by de Alfaro and Henzinger [AH01a]. Not considering input/output simplifies the quotient because a significantly simpler composition operator is involved. In addition, Beneš et al. assume a single global alphabet and do not consider alphabet extension, which is particularly difficult for the quotient, as seen above.

3.4.3. Computational Complexity of EMIA Operations

In this section we briefly discuss the computational complexity of the EMIA operators, assuming that all involved EMIAs are finite. Most of the operators are of polynomial complexity. In these cases, we give a concrete upper bound $\#(t)$ on the complexity of computing a term t. Sometimes we decompose our complexity consideration action-wise, where $\#^\alpha(t)$ stands for the complexity of computing t for a single action α. The cardinality of a set X is denoted $|X|$. Given an EMIA with state set S, a state $s \in S$ and a subset $Q \subseteq S$, we write T_s^α for the set of all α-transitions leaving s and T_Q^α for the set of all α-transitions leaving Q. Given a transition $t \in \mathrm{T}_s^\alpha$, we write $\mathrm{trg}(t)$ for the number of target states of t, i.e., if $t = s \xrightarrow{\alpha} S'$, then $\mathrm{trg}(t) = |S'|$. The total number of target states over all α-transitions of an EMIA R is

$$\mathrm{trg}_R^\alpha := \sum_{r \in S_R} \sum_{t' \in \mathrm{T}_r^\alpha} \mathrm{trg}(t') \tag{3.1}$$

Of course, the complexity of an operation depends on the data structure used to represent EMIAs. In the following, we assume a data structure that is optimal for the operation under consideration. However, a data structure optimal for one operation may be suboptimal for a different one. Therefore, the complexities given in this section may be unachievable in practice.

Refinement

The complexity of refinement checking has been studied for several variants of MTS in [Ben+11a]. According to these findings, the complexity is P-complete for both MTS and DMTS. Because EMIA is based on dMTS, these complexities constitute a lower and an upper bound on the complexity of checking strong refinement for EMIA, i.e., when not considering weak transitions. In the worst case, one has to touch all transitions in

order to calculate the weak extensions of a single transition. Hence, weakening adds at most quadratic complexity. Therefore, refinement checking on EMIAs is also P-complete.

Backwards Closure, Pruning and Galois Abstraction

In the worst case of computing the backwards closure $\mathrm{bcl}_Q^L(X)$ of a set of states $X \subseteq S_Q$ with respect to a set of actions $L \subseteq A_Q$ (Definition 3.39), one has to touch each state and each of its incoming transitions, resulting in the complexity

$$\#(\mathrm{bcl}_Q(X)) = |S_Q| \cdot \sum_{\alpha \in A} |\,\mathrm{T}_Q^\alpha\,|. \tag{3.2}$$

Because one may remove states on the fly while computing the backwards closure, bcl is the main source of complexity for the pruning operations employed in the Galois abstraction and the MIA parallel composition, as well as in conjunction and quotienting.

Parallel Product and Pre-quotient

The same consideration applies to the parallel product and the pre-quotient. Therefore, we only elaborate for the parallel product.

In order to calculate the parallel product $Q \otimes R$ of two EMIAs Q and R, we iterate, for each pair of states $(q, r) \in S_Q \times S_R$ and each action α, through all pairs of α-transitions of q and r. Hence, the complexity of the local product $q \otimes r$ for a single action α is

$$\#^\alpha(q \otimes r) = \sum_{t \in \mathrm{T}_q^\alpha} \sum_{t' \in \mathrm{T}_r^\alpha} \mathrm{trg}(t) \cdot \mathrm{trg}(t'). \tag{3.3}$$

The complexity of $Q \otimes R$ with respect to α is then

$$\begin{aligned}
\#^\alpha(Q \otimes R) &= \sum_{q \in S_Q} \sum_{r \in S_R} \#^\alpha(q \otimes r) && && (3.4)\\
&= \sum_{q \in S_Q} \sum_{r \in S_R} \sum_{t \in \mathrm{T}_q^\alpha} \sum_{t' \in \mathrm{T}_r^\alpha} \mathrm{trg}(t) \cdot \mathrm{trg}(t') && \text{by (3.3)} && (3.5)\\
&= \sum_{q \in S_Q} \sum_{t \in \mathrm{T}_q^\alpha} \sum_{r \in S_R} \sum_{t' \in \mathrm{T}_r^\alpha} \mathrm{trg}(t) \cdot \mathrm{trg}(t') && \text{by commutativity} && (3.6)\\
&= \left(\sum_{q \in S_Q} \sum_{t \in \mathrm{T}_q^\alpha} \mathrm{trg}(t) \right) \cdot \left(\sum_{r \in S_R} \sum_{t' \in \mathrm{T}_r^\alpha} \mathrm{trg}(t') \right) && \text{by distributivity} && (3.7)\\
&= \mathrm{trg}_Q^\alpha \cdot \mathrm{trg}_R^\alpha\,. && \text{by (3.1)} && (3.8)
\end{aligned}$$

Note that this complexity does not depend on the number of states but on the number of transitions and their target states. Hence, computing $Q \otimes R$ via the state space product $S_Q \times S_R$ may be particularly inefficient in cases with many unreachable states. If the

average numbers of target states $\overline{\mathrm{trg}}^{\alpha}_{Q}$ and $\overline{\mathrm{trg}}^{\alpha}_{R}$ are known, we can reformulate (3.8) as

$$\#^{\alpha}(Q \otimes R) = |S_Q| \cdot |S_R| \cdot \overline{\mathrm{trg}}^{\alpha}_{Q} \cdot \overline{\mathrm{trg}}^{\alpha}_{R}, \tag{3.9}$$

which captures the intuition that the complexity depends on the number of states and the average branching size. The overall complexity of the parallel product may be computed as

$$\#(Q \otimes R) = \sum_{\alpha \in A} \mathrm{trg}^{\alpha}_{Q} \cdot \mathrm{trg}^{\alpha}_{R} = |A| \cdot |S_Q| \cdot |S_R| \cdot \overline{\mathrm{trg}}_{Q} \cdot \overline{\mathrm{trg}}_{R}. \tag{3.10}$$

We do not need to consider unshared actions explicitly because a foreign action α corresponds to an α-must loop at each state, which yields $\mathrm{trg}^{\alpha}_{Q} = |S_Q|$. This result corresponds to the intuition that, in order to extend Q with loops, we have to touch each state once.

Hiding and Restriction

Given a set $L \subset A$ of actions of an EMIA Q, the worst case complexity of hiding and restriction is obviously

$$\#(Q \,/\, L) = \#(Q \setminus L) = \sum_{\alpha \in L} \mathrm{T}^{\alpha}_{Q}\,. \tag{3.11}$$

Conjunctive Product

When computing the conjunction $Q \,\&\, R$ of two EMIAs Q and R, one combines any α-transition of a state $q \in S_Q$ with all α-may-transitions of a state $r \in S_R$ and vice versa. We get

$$\#^{\alpha}(q \,\&\, r) = \sum_{t \in \mathrm{T}^{\alpha}_{q}} \mathrm{trg}^{\alpha}(t) \cdot |\,\mathrm{may}^{\alpha}_{R}(r)| + \sum_{t \in \mathrm{T}^{\alpha}_{r}} \mathrm{trg}^{\alpha}(t) \cdot |\,\mathrm{may}^{\alpha}_{Q}(q)|. \tag{3.12}$$

The complexity of $Q \,\&\, R$ with respect to α is then

$$\#^{\alpha}(Q \,\&\, R) = \sum_{q \in S_q} \sum_{r \in S_R} \#^{\alpha}(q \,\&\, r) \tag{3.13}$$

$$= \sum_{q \in S_q} \sum_{r \in S_R} \left(\sum_{t \in \mathrm{T}^{\alpha}_{q}} \mathrm{trg}^{\alpha}(t) \cdot |\,\mathrm{may}^{\alpha}_{R}(r)| + \sum_{t' \in \mathrm{T}^{\alpha}_{r}} \mathrm{trg}^{\alpha}(t') \cdot |\,\mathrm{may}^{\alpha}_{Q}(q)| \right) \tag{3.14}$$

$$= \left(\sum_{q \in S_q} \sum_{r \in S_R} \sum_{t \in \mathrm{T}^{\alpha}_{q}} \mathrm{trg}^{\alpha}(t) \cdot |\,\mathrm{may}^{\alpha}_{R}(r)| \right) + \tag{3.15}$$

$$\left(\sum_{q \in S_q} \sum_{r \in S_R} \sum_{t' \in \mathrm{T}^{\alpha}_{r}} \mathrm{trg}^{\alpha}(t') \cdot |\,\mathrm{may}^{\alpha}_{Q}(q)| \right) \tag{3.16}$$

$$= \left(\sum_{q \in S_q} \sum_{t \in \mathrm{T}_q^\alpha} \mathrm{trg}^\alpha(t) \right) \cdot \sum_{r \in S_R} |\,\mathrm{may}_R^\alpha(r)| \quad + \tag{3.17}$$

$$\left(\sum_{r \in S_R} \sum_{t' \in \mathrm{T}_r^\alpha} \mathrm{trg}^\alpha(t') \right) \cdot \sum_{q \in S_q} |\,\mathrm{may}_Q^\alpha(q)| \tag{3.18}$$

$$= \mathrm{trg}_Q^\alpha \cdot |\,\mathrm{may}_R^\alpha\,| + \mathrm{trg}_R^\alpha \cdot |\,\mathrm{may}_Q^\alpha\,|. \tag{3.19}$$

As a consequence, the overall complexity of $Q \,\&\, R$ is

$$\#(Q \,\&\, R) = \sum_{\alpha \in A} \mathrm{trg}_Q^\alpha \cdot |\,\mathrm{may}_R^\alpha\,| + \mathrm{trg}_R^\alpha \cdot |\,\mathrm{may}_Q^\alpha\,|. \tag{3.20}$$

In terms of the state spaces and the average sizes $\overline{\mathrm{trg}}_Q$, $\overline{\mathrm{trg}}_R$, $\overline{\mathrm{may}}_Q$ and $\overline{\mathrm{may}}_R$, 3.20 may be reformulated as

$$\#(Q \,\&\, R) = |A| \cdot |S_Q| \cdot |S_R| \cdot (\overline{\mathrm{trg}}_Q \cdot \overline{\mathrm{may}}_R + \overline{\mathrm{trg}}_R \cdot \overline{\mathrm{may}}_Q). \tag{3.21}$$

Disjunction

Because disjunction corresponds to a disjoint union, its complexity depends on whether the operation happens in-place or out-of-place. An in-place implementation obviously has constant complexity for joining the initial states. For an out-of-place implementation, one additionally needs to copy the two components, i.e., to touch each state and each transition. Hence,

$$\#(Q \sqcup R) = |Q| + |R| + \mathrm{trg}_Q + \mathrm{trg}_R\,. \tag{3.22}$$

3.4.4. Temporal Logic Integration

We have developed EMIA into a heterogeneous specification theory in the sense of Section 2.1 in so far as it supports state-based specification based on modal transition systems and logic-based specification by providing a conjunction and a disjunction operator on EMIAs. In addition, we would appreciate to support temporal logic formulas in the style of HML, but on the level of EMIAs, e.g., given an EMIA P and an action a, we would like to express a formula $\phi := [a]P$ (see Section 2.2.5) as an EMIA M_ϕ in such a way that logical satisfaction corresponds to refinement:

$$P \vDash \phi \iff P \sqsubseteq M_\phi. \tag{3.23}$$

This would allow for truly heterogeneous specification, where transition systems and logical formulas may be freely intermixed. In their work on Logic LTS, Lüttgen and Vogler elaborate on this idea for LTS and MTS, however within a different semantic setting based on ready-simulation and a CSP-style parallel operator [LV11]. This work has been adapted to MIA by Bujtor and Vogler [BV16]. An adaptation to EMIA should

be straightforward, including for the Galois insertion. However, we do not anticipate much new insight from such an adaptation and, therefore, leave it for future work.

4. Case Studies and Tool Support

In this chapter we summarise the incremental, component-based design methodology provided by interface theories and illustrate it by means of two case studies. In the first study on a client-server system, we design a nondeterministic system using MIA. Although designed manually, all examples of this study are checked with tool support. In the second case study, we employ EMIA to design a railway control system sufficiently large to require tool support. We also present our two tool implementations Gemia and MiaGo that have been developed during the work on this thesis and have been used to carry out these studies.

4.1. The Incremental, Component-based Design Methodology

The incremental, component-based design methodology provided by interface theories is mainly based on the compositional refinement preorder and the operators parallel composition, quotienting and conjunction. The refinement preorder allows a systems designer to start with a coarse specification and then refine it step-by-step by taking decisions about underspecified behaviour. As soon as a system is fully specified, i.e., any underspecification is resolved, one reaches an implementation represented as a labelled transition system.

Due to the compositionality of the refinement preorder, this design methodology may be employed for designing system components independently and composing the overall system from these components via the parallel composition operator. Such a composition may be checked for the mutual communication requirements of its components. Potential incompatibilities may be resolved by suitably refining the components or the system. The parallel operator allows one to design a system incrementally by starting with a subset of the components to which further components may be added successively. This approach is further supported by the quotient operator, which is adjoint to parallel composition and enables component reuse by synthesising a specification of the remaining part(s) of a system from the reused component and a global system specification.

The conjunction operator permits perspective-based specification where one may specify several requirements on a component or a system independently. The overall requirement is obtained as the conjunction of the individual requirements.

4.2. Designing a Client Server System with MIA

In this section we discuss a case study that employs MIA to design a client-server system [Buj+16]. The example exercises several important operators of MIA; it also

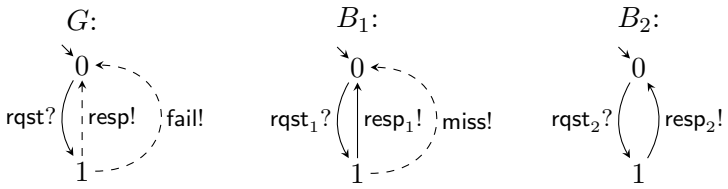


Figure 4.1.: Global specification G, local cache B_1 and remote database B_2 with the alphabets $A_G = \{\mathsf{rqst}\}/\{\mathsf{resp}, \mathsf{fail}\}$, $A_{B_1} = \{\mathsf{rqst}_1\}/\{\mathsf{resp}_1, \mathsf{miss}\}$ and $A_{B_2} = \{\mathsf{rqst}_2\}/\{\mathsf{resp}_2\}$.

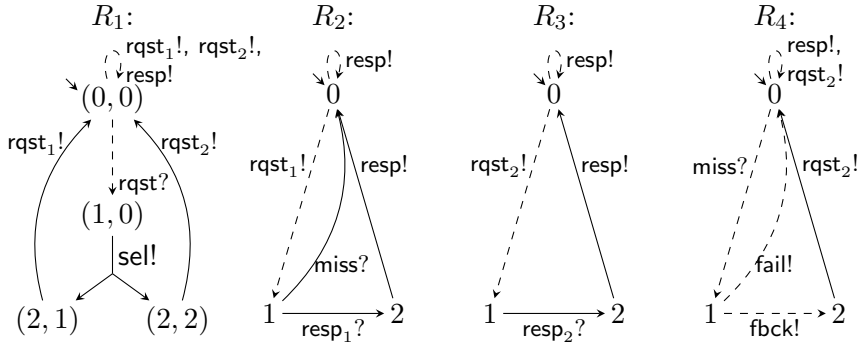


Figure 4.2.: Front-end requirements R_1 through R_4 with the alphabets $A_1 = \{\mathsf{rqst}\}/\{\mathsf{rqst}_1, \mathsf{rqst}_2, \mathsf{resp}, \mathsf{sel}\}$, $A_2 = \{\mathsf{resp}_1, \mathsf{miss}\}/\{\mathsf{rqst}_1, \mathsf{resp}\}$, $A_3 = \{\mathsf{resp}_2\}/\{\mathsf{rqst}_2, \mathsf{resp}\}$ and $A_4 = \{\mathsf{miss}\}/\{\mathsf{rqst}_2, \mathsf{resp}, \mathsf{fail}, \mathsf{fbck}\}$.

uses nondeterminism, which means that it cannot be modelled in MI [Rac+11]. All constructions have been checked with our Haskell implementation of MIA, which we present in Section 4.4.1. The source code of our example is shown in Appendix C.1 and available online [Fen17]. Although the results are generated automatically, we improved the layout of the figures manually.

We consider a data server S that is composed of a front-end F and two already existing back-ends, a local cache B_1 and a remote database B_2. The server forwards requests received by the front-end to (one of) the two back-ends. Based on a global specification G of S, we wish to develop the specification of F. The global specification and the back-end specifications are shown in Figure 4.1.

Specification G defines the communication protocol with a client. The data server shall wait for a request and then may return a response or, alternatively, a failure message. Action rqst? is of must-modality because a data server makes no sense if it cannot accept a request. Actions resp! and fail! are of may-modality since refinements of G might at some stage decide to give only answer resp! or only fail!. The local cache B_1 also waits for a request and answers with a response; optionally, it may implement a cache miss after a request (miss). The remote database B_2 is similar to the cache but without a miss. In both cases we have must-transitions for $\mathsf{rqst}_i?$ and $\mathsf{resp}_i!$, so that the acceptance of inputs and issuing of answers is guaranteed.

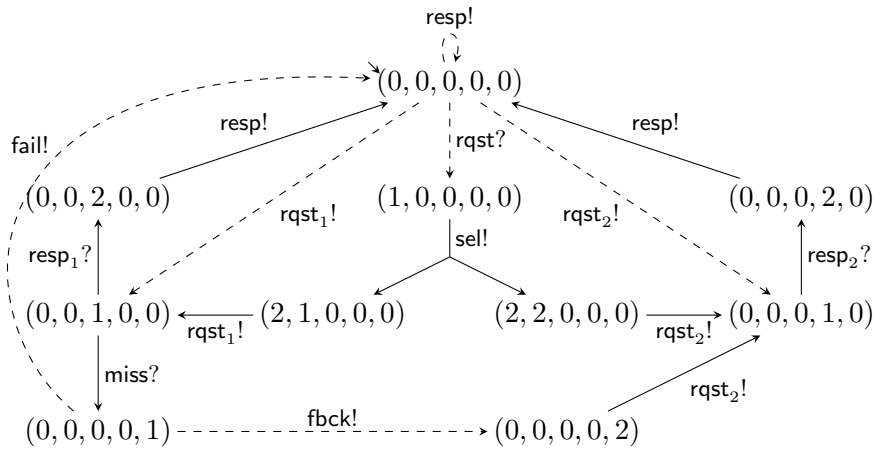


Figure 4.3.: Conjunction of the front-end requirements, $R = R_1 \sqcap R_2 \sqcap R_3 \sqcap R_4$ with the alphabets $I = \{\mathsf{rqst}, \mathsf{resp}_1, \mathsf{resp}_2, \mathsf{miss}\}$, $O = \{\mathsf{resp}, \mathsf{rqst}_1, \mathsf{rqst}_2, \mathsf{sel}, \mathsf{fbck}, \mathsf{fail}\}$ (implicitly extended).

We now develop the front-end specification F, which forwards a request to either cache B_1 or to database B_2. In case of the former and a cache miss, F may fall back to B_2. To this end, we assume the following requirements for F, which are specified in Figure 4.2:

R1. The front-end shall pass on a client's request to one of the back-ends.
R2. After forwarding a request to back-end B_1, the front-end shall wait for B_1's response and route it back to the client. Additional to the response, the front-end shall accept a cache miss when waiting for a response.
R3. After redirecting the request to back-end B_2, the front-end shall wait for B_2's response and route it back to the client.
R4. In case of a cache miss, the front-end may fall back to the database or fail.

Requirement R_1 specifies that after receiving a request ($\mathsf{rqst}?$), a back-end is selected ($\mathsf{sel}!$) to which the request has to be forwarded ($\mathsf{rqst}_1!$, $\mathsf{rqst}_2!$). Requirement R_2 states that, after forwarding a request to the cache ($\mathsf{rqst}_1!$), the front-end must wait for a response ($\mathsf{resp}_1?$) or a cache miss ($\mathsf{miss}?$). In case of a response ($\mathsf{resp}_1?$), the response has to be routed back to the client ($\mathsf{resp}!$). Requirement R_3 is the corresponding requirement for the database back-end. Requirement R_4 specifies that, in case of a cache miss, the request can be redirected to the database back-end ($\mathsf{fbck}!$) or the whole conversation may fail ($\mathsf{fail}!$).

The conjunction $R := R_1 \sqcap R_2 \sqcap R_3 \sqcap R_4$ is shown in Figure 4.3, where the alphabets of the conjuncts are extended implicitly, and inconsistent and unreachable states are pruned. Observe that one could simplify R by merging states $(0, 0, 0, 0, 2) \equiv_{\mathrm{m}} (2, 2, 0, 0, 0)$.

All in all, the desired front-end specification F must guarantee that (i) the server S obeys the global specification, (ii) S is the parallel composition of the front-end and the

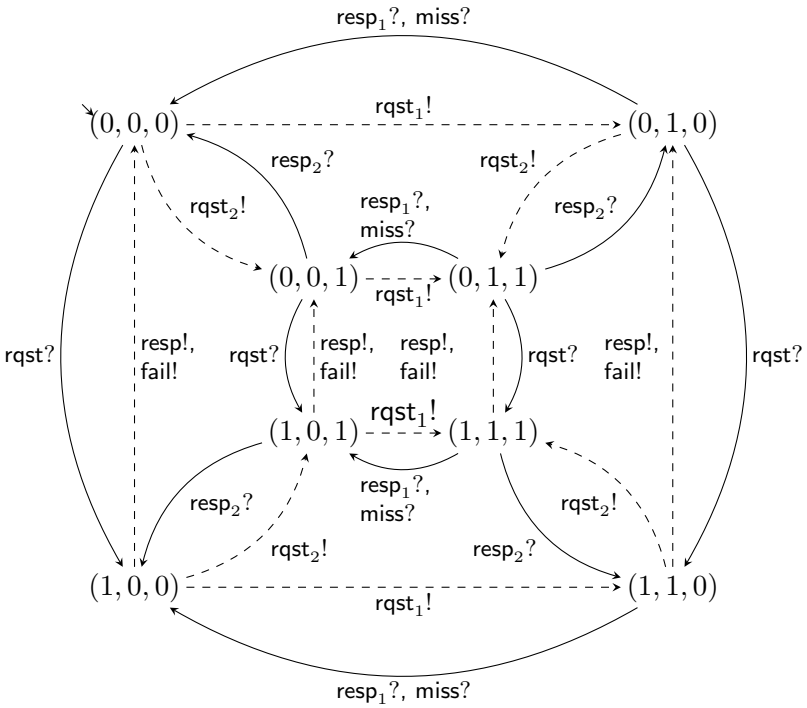


Figure 4.4.: Upper bound U_F on F with the alphabets $I := \{\mathsf{rqst}, \mathsf{resp}_1, \mathsf{resp}_2, \mathsf{miss}\}$ and $O := \{\mathsf{resp}, \mathsf{rqst}_1, \mathsf{rqst}_2, \mathsf{fail}\}$.

two back-ends, and (iii) F satisfies all its requirements. Formally:

$$S \sqsubseteq_{\mathrm{m}} G \qquad\qquad S \equiv_{\mathrm{m}} F \parallel B_1 \parallel B_2 \qquad\qquad F \sqsubseteq_{\mathrm{m}} R$$

Quotienting now gives us an upper bound U_F on F. To satisfy the alphabet requirements for quotienting, we first need to extend G's alphabet with the unknown actions $O' := \{\mathsf{rqst}_1, \mathsf{rqst}_2, \mathsf{resp}_1, \mathsf{resp}_2, \mathsf{miss}\}$ of $B_1 \parallel B_2$; see the discussion at the end of Section 3.1.9 and observe that these actions are indeed outputs in the parallel composition of F with $B_1 \parallel B_2$. Now,

$$U_F = [G]_{\emptyset,O'} \mathbin{/\!\!/} (B_1 \parallel B_2)\,,$$

i.e., U_F (see Figure 4.4) is the least specific interface that composes with the back-ends such that, after hiding of O', they together satisfy the global specification, as discussed in Section 3.1.9. Hence, overall:

$$(U_F \parallel B_1 \parallel B_2) \,/\, O' \sqsubseteq_{\mathrm{m}} G.$$

Note that, in Figure 4.4, we have omitted the universal state and its transitions, labelled resp_1, resp_2 and miss. These transitions do not play a role in $U_F \sqcap R$ in the next step, because the only three transitions in R with these labels are solely combined with transitions actually shown in Figure 4.4.

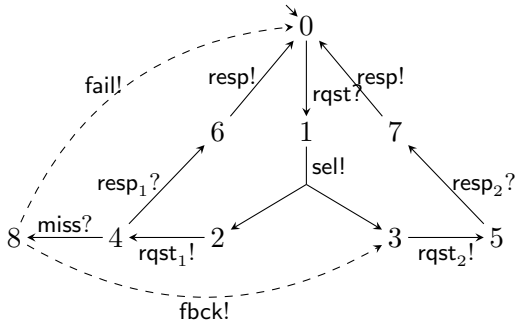


Figure 4.5.: Final front-end specification F with alphabets $I = \{\mathsf{rqst}, \mathsf{resp}_1, \mathsf{resp}_2, \mathsf{miss}\}$ and $O = \{\mathsf{rqst}_1, \mathsf{rqst}_2, \mathsf{resp}, \mathsf{sel}, \mathsf{fail}, \mathsf{fbck}\}$.

Thus, the front-end is specified as follows, because it also has to satisfy the requirements given by R:

$$F := U_F \sqcap R$$

This specification leaves the implementor as much freedom as possible. It is shown in Figure 4.5, where all unreachable and inconsistent states have already been removed.

This case study illustrates the usage of the operators parallel composition, conjunction and quotienting for designing a nondeterministic system that requires disjunctive must-transitions. The example demonstrates perspective-based specification by combining requirements R_1 through R_4 conjunctively and shows how to design a system incrementally by extending a partial implementation via quotienting. Also, the quotient in this example has must-transitions.

4.3. Designing a Railway Control System with EMIA

This section describes a case study involving a railway control system. The goal is to demonstrate the practical applicability of the EMIA interface theory with an example that requires tool support because it is too large to be designed manually. However, we omit many details that are relevant for a real railway control system in order to make the example easy to understand for a broad audience. All examples have been carried out with our tool MiaGo, the visualisations are generated automatically using Graphviz [Gra17].

We start by applying stepwise refinement to the specification of a single component (Section 4.3.1) and illustrate the composition of a small system from two such components (Section 4.3.2). The same procedure is carried out on variants of these components, in order to illustrate the differences when employing fatal error states (Section 4.3.3). The actual case study of a railway control system is then presented in Section 4.3.4.

In the automatically generated visualisations, we employ the following node shapes (Figure 4.6). An initial state is represented by a rectangle with an incoming, source-less arrow, ordinary states as elliptic nodes universal states are filled in yellow, and fatal error states are indicated by a diamond shape filled in red.

Figure 4.6.: Illustration of initial, ordinary, universal and erroneous states.

Figure 4.7.: Initial signal design (left) and extended alphabet (right).

4.3.1. Designing a Single Component

We illustrate the incremental design process of a single component by means of the specification of a railway signal. One may always start the design process with a fully underspecified interface, i.e., an EMIA consisting of an empty alphabet and a single universal state as initial state (Figure 4.7 (left)), and refine this universal specification step by step.

The first step is to extend the alphabet by necessary actions. In case of the railway signal, we want to switch the signal between a stop and a go mode; hence, we extend the alphabet by two input actions stop and go (Figure 4.7 (right)).

In order to have an observable effect, the signal also needs two states s and g representing the stop and the go mode (Figure 4.8 (left)), respectively. We decide that the signal always initialises itself in the stop state s and that action go transfers to state g, from which action stop transfers back to s. The cases of receiving a stop signal in state s or a go signal in state g are left fully underspecified in order to defer the decision to the next refinement steps.

Next, we want to refine the underspecification of action go in state g and of action stop in state s, which currently allow for arbitrary behaviour. An adequate reaction to such a signal would be either to ignore it or to consider it as an error. These options are collected using the disjunctive must-transitions in Figure 4.8 (right).

4.3.2. Combining Components

In principle, there are two interesting implementations of the above specification. Either (A) the signal ignores receiving an action multiple times, or (B) the signal considers an

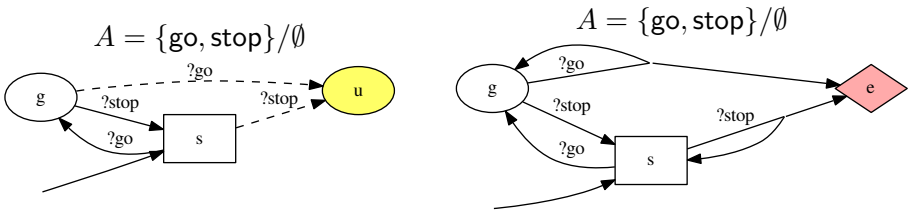


Figure 4.8.: Refinements of the signal design.

Figure 4.9.: Refinements S_0 and, respectively, S_1 of the signal design (A) up to renaming of actions.

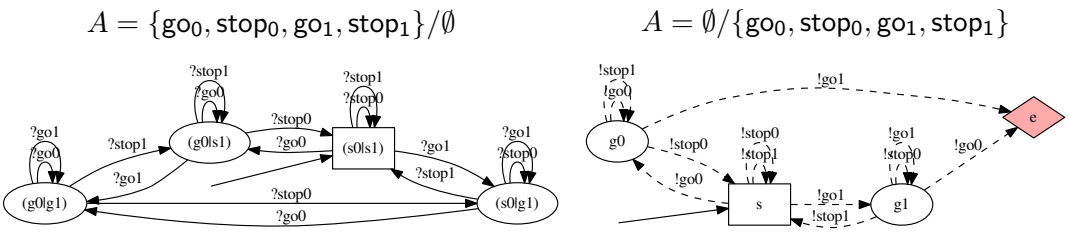


Figure 4.10.: Composition $S_1 \otimes S_2$ of two signals of type (A) (left) and safety requirement R_C on the controller (right).

action as an error when receiving it several times. Of course one may also combine these behaviours, but such a combination does not give us additional insight.

We start by investigating option (A) for two signals S_0 and S_1 that shall secure the junction of two railway tracks (Figure 4.9). Composing these two specifications in parallel reveals the behaviour $S_1 \otimes S_2$ that the signals may exhibit in principle (Figure 4.10 (left)). Of course, the two signals must not be in their respective go states simultaneously. We express this as a requirement on the controller by the specification R_C shown in Figure 4.10 (right).

Note that in error-abstracting interface theories such as IA, MI or MIA, one cannot employ an error state in order to forbid output transitions, because the error would immediately be propagated backwards to the initial state although the specification has error-free implementations. Alternatively, one may remove the output may-transitions leading to the error state. This way, one would simultaneously express that error states should be unreachable, which we include anyway as a global requirement in the next step. However, employing an error state makes the purpose of the requirement more explicit. In particular, it eases the specification of more complex requirements including must-transitions. For example, if we had a $!\mathsf{go}_1$-must-transition from g_0 to e, then we would have to remove the $!\mathsf{go}_0$-may-transition from s to g_0. This backward propagation of error information along output must-transitions happens automatically when employing error states and a suitable global requirement.

The global requirement G that we use in order to synthesise a safe controller permits any output sequence in general (Figure 4.11). However, due to the absence of an error state, the system is required to be error-free. Note that the state of G is *not* universal.

$$A = \emptyset/\{\mathsf{go}_0, \mathsf{stop}_0, \mathsf{go}_1, \mathsf{stop}_1\}$$

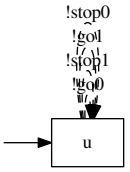


Figure 4.11.: Global specification G requiring error-freedom. G includes a may-loop for each action.

$$A = \emptyset/\{\mathsf{go}_0, \mathsf{stop}_0, \mathsf{go}_1, \mathsf{stop}_1\}$$

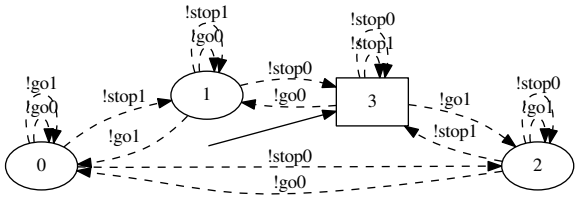


Figure 4.12.: Quotient $G /\!\!/ (S_0 \otimes S_1)$ for signal type (A).

A safe controller C must satisfy the following inequations:

$$C \sqsubseteq R_C, \tag{4.1}$$

$$C \otimes S_0 \otimes S_1 \sqsubseteq G. \tag{4.2}$$

As a consequence of (4.2) we get $C \sqsubseteq G /\!\!/ (S_0 \otimes S_1)$, leading to the overall inequation

$$C \sqsubseteq R_C \sqcap (G /\!\!/ (S_0 \otimes S_1)). \tag{4.3}$$

Figure 4.12 illustrates the quotient in (4.3) and Figure 4.13 its conjunction with R_C. Note that there are many unreachable states shown in the conjunction, leaving only three reachable states. This specification gives an upper bound on the controller that guarantees safe operation of the railway junction. It is clear that no implementation of this specification may guide the system into an unsafe state.

In this simple example, it was easy to specify the controller requirement R_C directly. In a more complex situation, it may be easier to identify unwanted behaviour in the parallel composition $S_1 \otimes S_2$. The safety requirement can then be specified as an observer that is obtained from $S_1 \otimes S_2$ by marking unwanted states as errors. Transitions leaving such a state may be removed. In our case, state $(g_0 \otimes g_1)$ has to be marked as erroneous (Figure 4.14).

Using global specification G, a safe controller C' must now satisfy the inequations

$$C' \otimes R_{C'} \sqsubseteq G, \tag{4.4}$$

$$C' \otimes S_0 \otimes S_1 \sqsubseteq G. \tag{4.5}$$

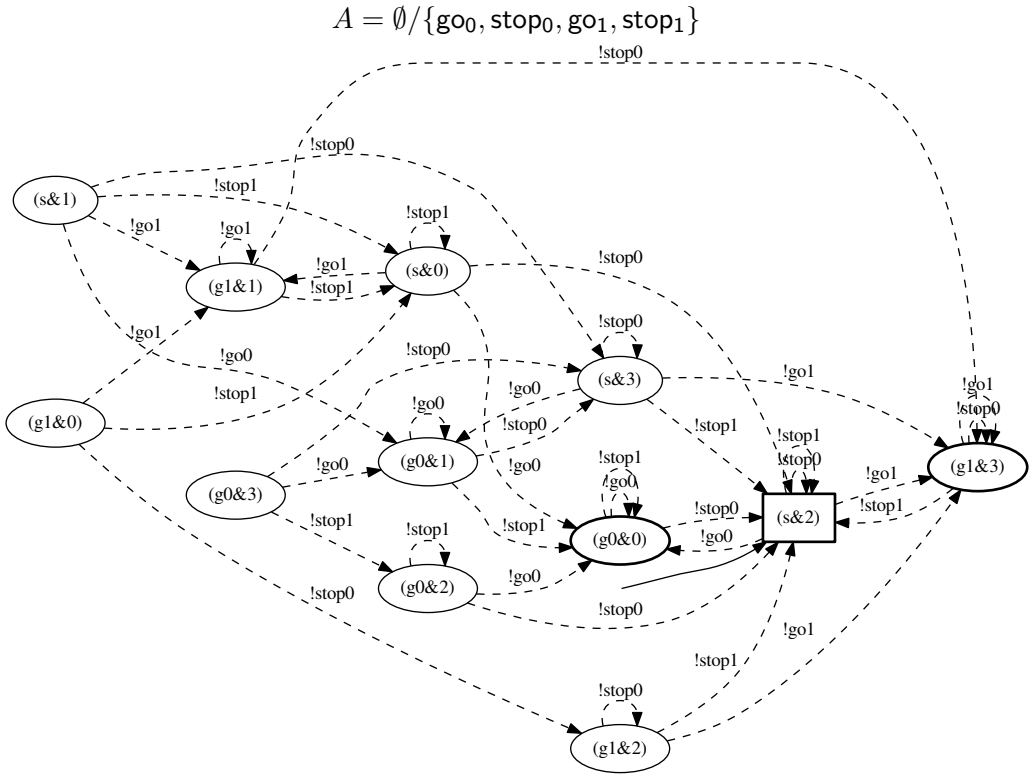


Figure 4.13.: Specification $R_C \sqcap (G /\!\!/ (S_0 \otimes S_1))$ of a safe controller with reachable states marked in bold.

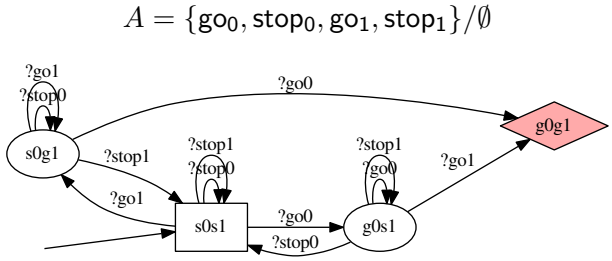


Figure 4.14.: Safety requirement $R_{C'}$ specified as an observer.

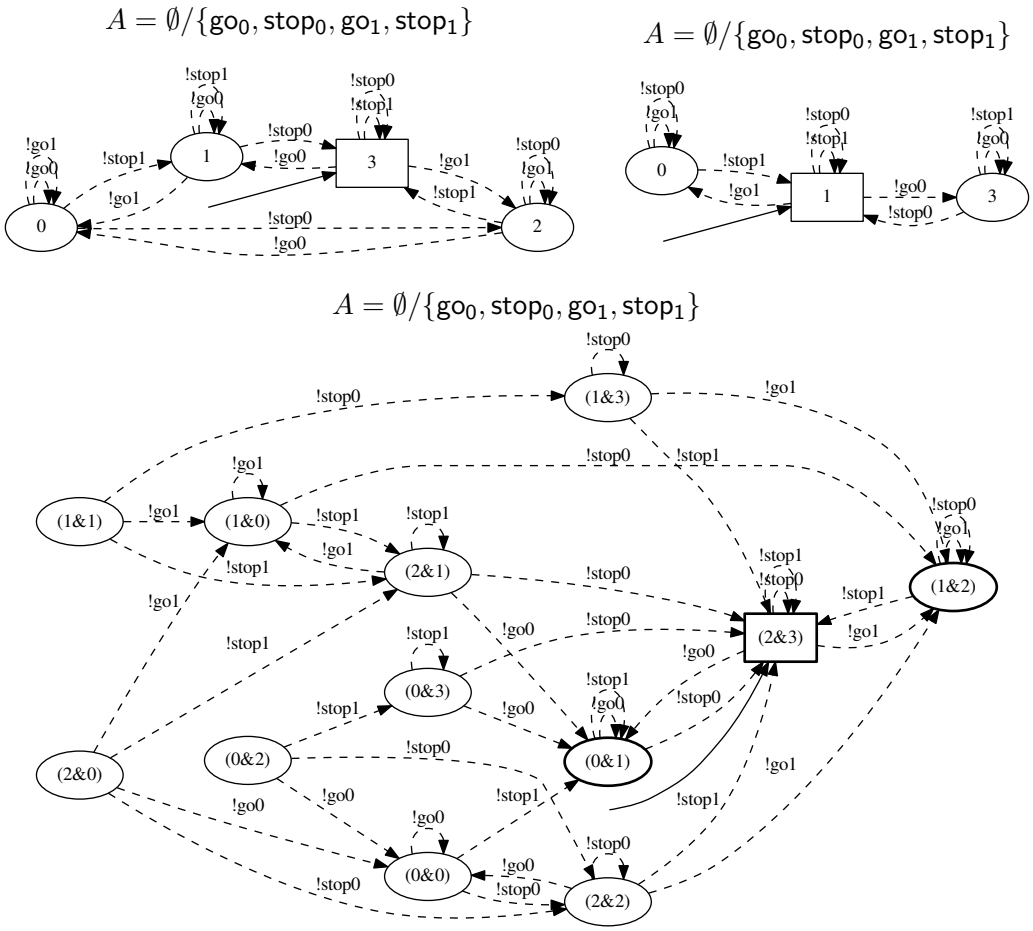


Figure 4.15.: Quotients $G /\!\!/ (S_0 \otimes S_1)$ (top left) and $G /\!\!/ R_{C'}$ (top right) and their conjunction (bottom, with reachable states marked in bold).

Hence, $C' \sqsubseteq G /\!\!/ R_{C'}$ and $C' \sqsubseteq G /\!\!/ (S_1 \otimes S_2)$ yielding the overall inequation

$$C' \sqsubseteq (G /\!\!/ (S_0 \otimes S_1)) \sqcap (G /\!\!/ R_{C'}). \tag{4.6}$$

Figure 4.15 illustrates the two quotients in (4.6) and their conjunction. Again, there are many unreachable states shown in the conjunction, leaving only the three reachable states $s\&3$, $g_0\&1$ and $g_1\&2$.

4.3.3. Variation of the Components

Here, we briefly discuss the variation of the above system when employing signals of type (B), which consider receiving signal stop in state s or signal go in state g as an error (Figure 4.16). Their parallel composition is shown in Figure 4.17.

Using the same safety requirement and the same global specification as above, the

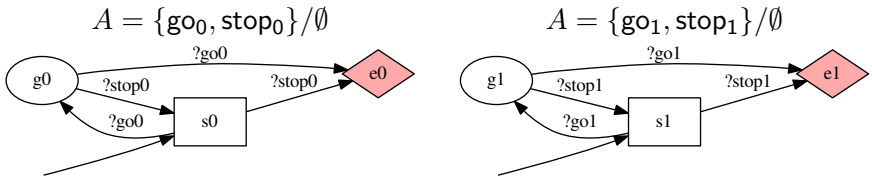


Figure 4.16.: Refinements of signal design (B).

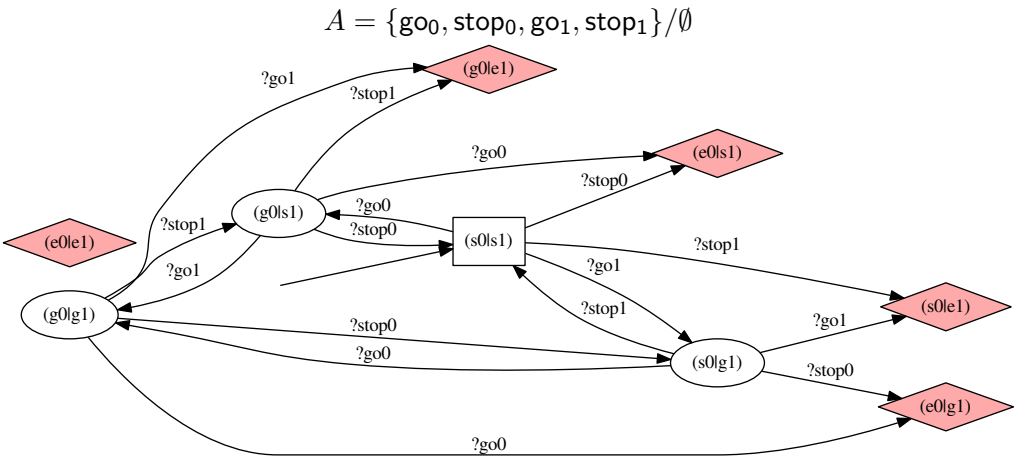


Figure 4.17.: Composition of two signals of type (B).

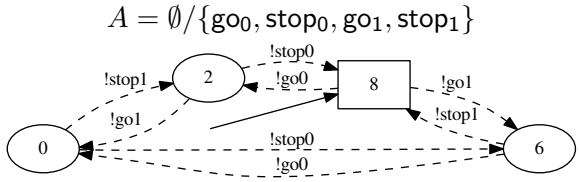


Figure 4.18.: Quotient $G /\!\!/ (S_0 \otimes S_1)$ for signal type (B).

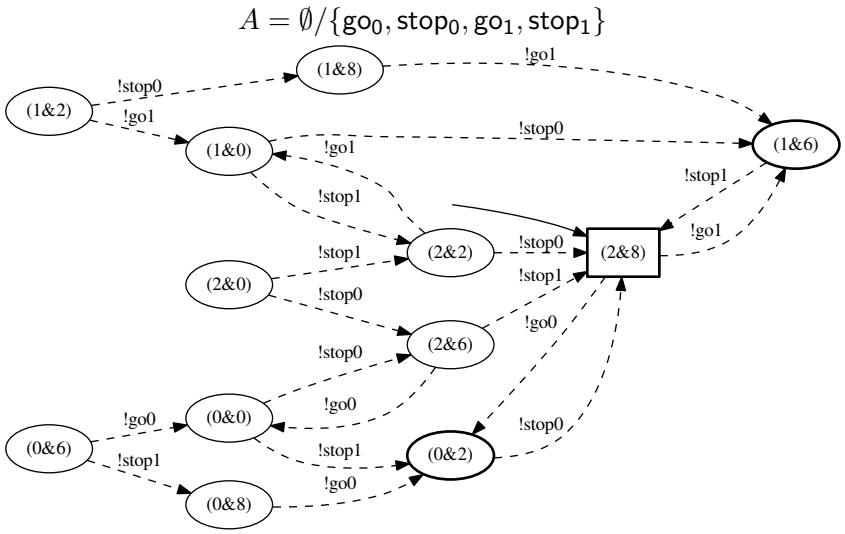


Figure 4.19.: Specification of the controller for signal type (B) with reachable states marked in bold.

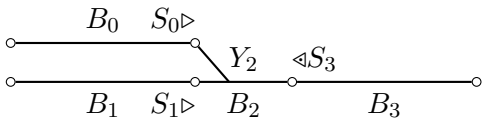


Figure 4.20.: Track layout of a safety-critical railway system.

quotient is shown in Figure 4.18. This leads to the controller shown in Figure 4.19, with again only three reachable states.

Remarkably, when comparing with variant (A), the may-loops disappeared at several states in the quotient and the controller, because these actions would provoke an error when employing signal variant (B).

4.3.4. Controlling a Railway Junction

In the previous section, we have illustrated the design process of a safety-critical system by means of an example that is small enough to be checked manually. In this section, we apply the same design process to a larger system that requires tool support. Again, we consider a railway system that has to be operated safely by a controller.

The track layout is illustrated in Figure 4.20. The track is divided into several blocks B_i that are bounded by small circles, where i denotes a unique block identifier. The purpose of these blocks is to organise a safe operation of trains on the track. A block may be either free or blocked, and one wishes to ensure that a block is blocked at most once in order to prevent trains from sharing the same block and risking an accident. The symbol $\triangleright$ indicates that the end of a block is secured by a signal S_i towards the right, i.e., a train coming from the left may have to stop at the signal in order to not endanger

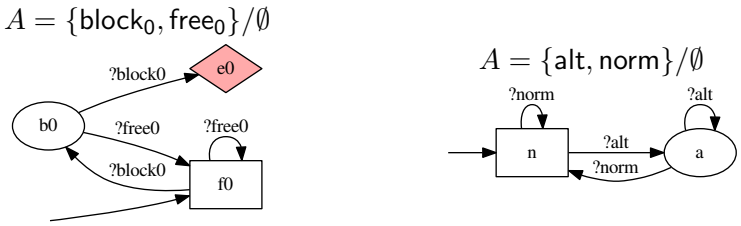


Figure 4.21.: Specification of Block B_0 (left) and Switch Y_2 (right).

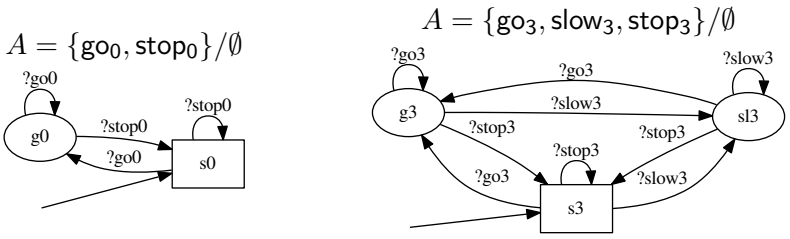


Figure 4.22.: Two-mode signal S_0 and three-mode signal S_3.

the next block to the right. Similarly, a signal indicated by $\triangleleft$ secures the end of a block towards the left (not used in the example of Figure 4.20). The symbols $\triangleleft$ and $\triangleright$ indicate three-mode signals which, in addition to the modes 'stop' and 'go', have a third mode 'slow'. We indicate switches by Y and assume that the normal direction of a switch is straight and the alternate direction is to turnoff. In summary, the basic components are blocks, two-mode signals, three-mode signals and switches.

Our track layout has four blocks, B_0 through B_3. An EMIA-specification of B_0 is shown in Figure 4.21 (left), the other blocks are specified analogously. Each block may be either free or blocked. When trying to block the same block multiple times, an error occurs. In this sense, 'blocked' means 'blocked by a different train'. Note that this treatment of blocks as binary values (blocked or free) simplifies reality significantly. In a real railway controlling system a block would be blocked by a train, which would require to send a train-ID when blocking a track section. We employ this simplification because we do not support the modelling of data in the current state of the theory and the tool implementation (see Chapter 5 for an extension). A switch may be in normal or in alternate position (Figure 4.21, right).

The track is equipped with two two-mode signals, S_0 and S_1, which allow or disallow a train to enter the junction from Blocks B_0 and B_1, respectively (Figure 4.22, left). Further, Three-mode Signal S_3 (Figure 4.22, right) controls trains coming from the right through B_3. The purpose of its slow-mode is to limit the speed of trains taking the alternate direction.

The behaviour of the railway system is defined by the following parallel composition:

$$S := S_0 \otimes S_1 \otimes S_3 \otimes B_0 \otimes B_1 \otimes B_2 \otimes B_3 \otimes Y_2. \tag{4.7}$$

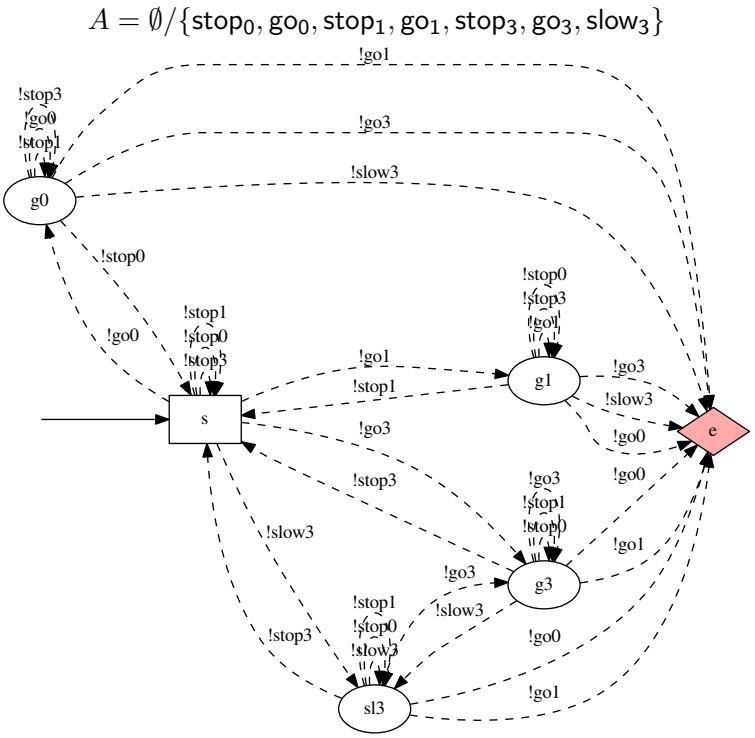


Figure 4.23.: Requirement R_1.

That is, S has $2 \cdot 2 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 2 = 2^3 \cdot 3^5 = 1944$ states. We consider the following requirements for the safe operation of this system:

R1. At most one of S_0, S_1 and S_3 may be in go or slow mode (Figure 4.23).
R2. If B_0 is blocked and Y_2 is in alternate position, then S_3 must be in stop mode (Figure 4.24).
R3. If B_1 is blocked and Y_2 is in normal position, then S_3 must be in stop mode (Figure 4.25).
R4. If B_2 is blocked, then S_0, S_1 and S_3 must be in their stop mode (Figure 4.26).
R5. If B_2 is blocked, then Y_2 may not change its state (Figure 4.27).
R6. If B_3 is blocked, then S_0 and S_1 must be in their stop mode (Figure 4.28).
R7. If S_0 is in go mode, then Y_2 must be in alternate position (Figure 4.29).
R8. If S_1 is in go mode, then Y_2 must be in normal position (Figure 4.30).
R9. If S_3 is in go mode, then Y_2 must be in normal position (Figure 4.30).

The controller's overall safety requirement is the following conjunction, for which we implicitly apply alphabet extension to each conjunct:

$$R_C := R_1 \sqcap \ldots \sqcap R_9. \tag{4.8}$$

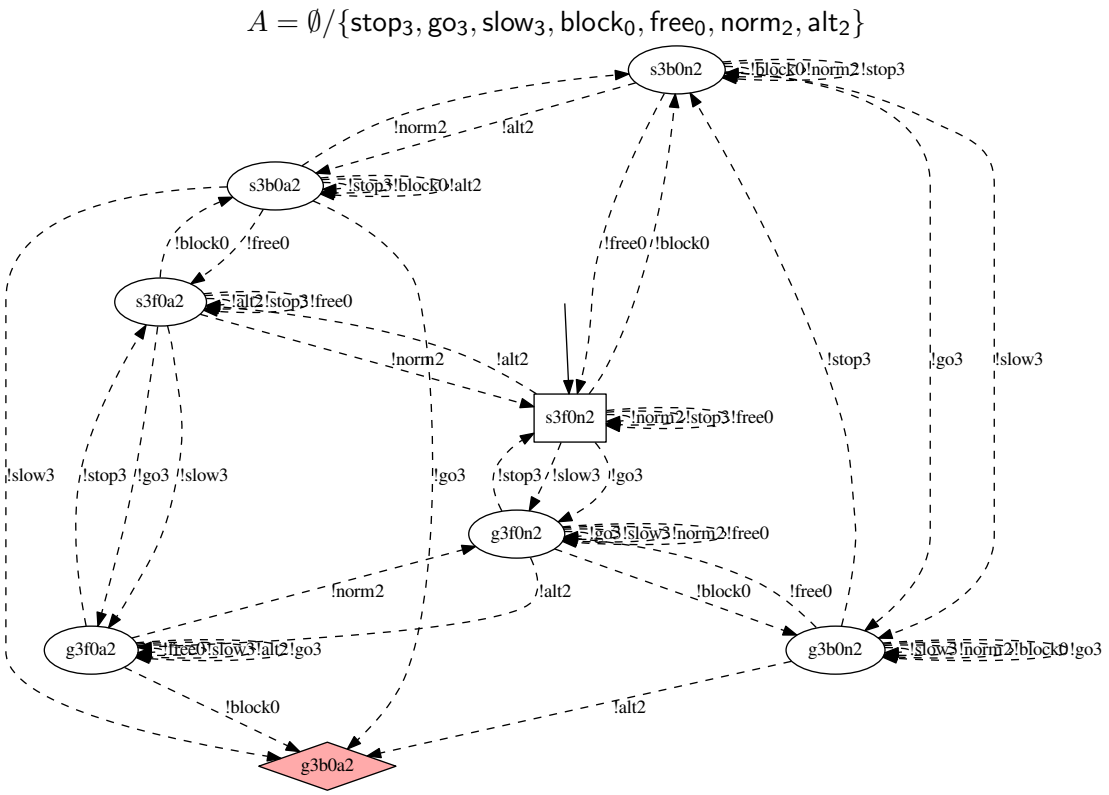


Figure 4.24.: Requirement R_2; the loops comprise those actions that do not change the state of the related component.

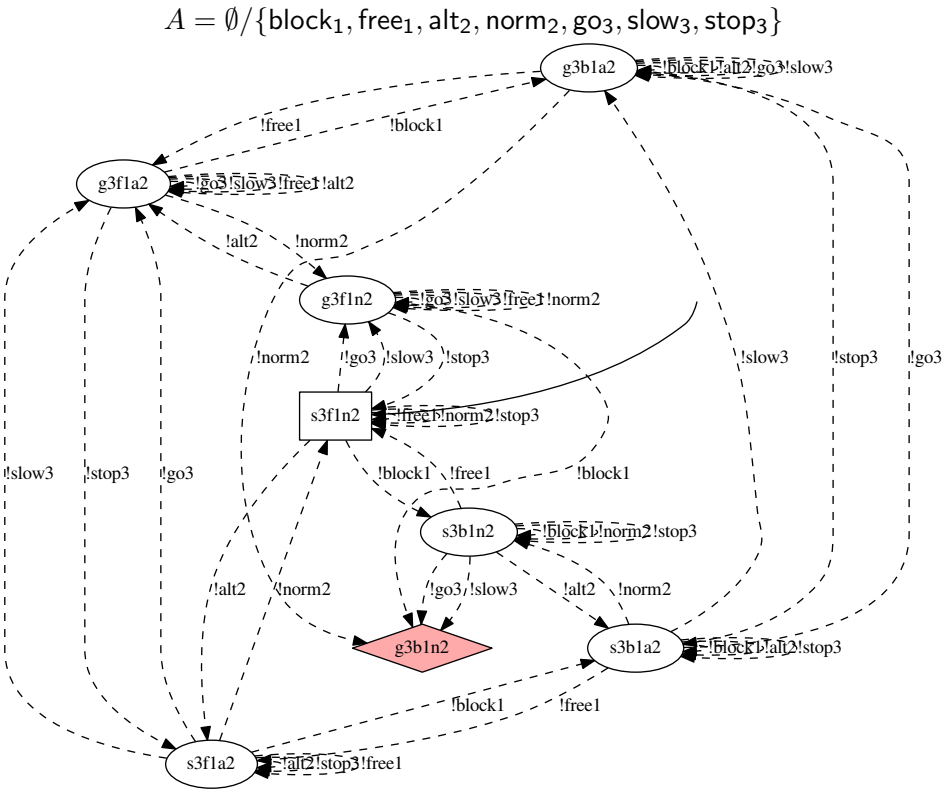
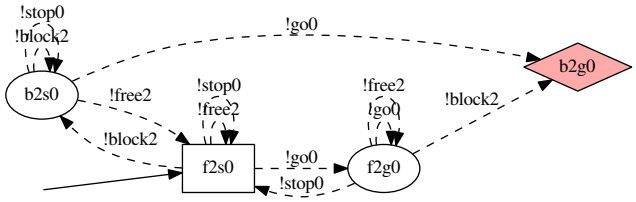
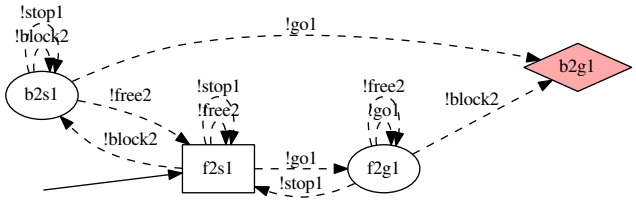


Figure 4.25.: Requirement R_3; the loops comprise those actions that do not change the state of the related component.

$A = \emptyset/\{\mathsf{go}_0, \mathsf{stop}_0, \mathsf{block}_2, \mathsf{free}_2\}$



$A = \emptyset/\{\mathsf{go}_1, \mathsf{stop}_1, \mathsf{block}_2, \mathsf{free}_2\}$



$A = \emptyset/\{\mathsf{block}_2, \mathsf{free}_2, \mathsf{go}_3, \mathsf{slow}_3, \mathsf{stop}_3\}$

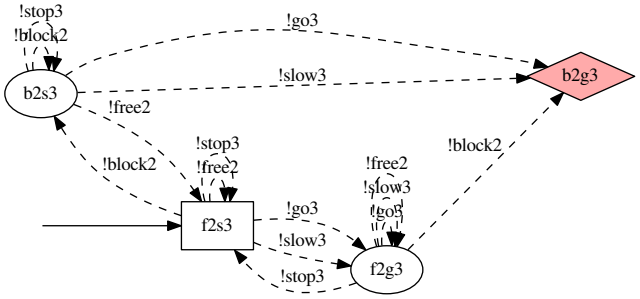


Figure 4.26.: Requirement R_4 results from a conjunction of three EMIAs.

$A = \emptyset/\{\mathsf{alt}_2, \mathsf{block}_2, \mathsf{free}_2, \mathsf{norm}_2\}$

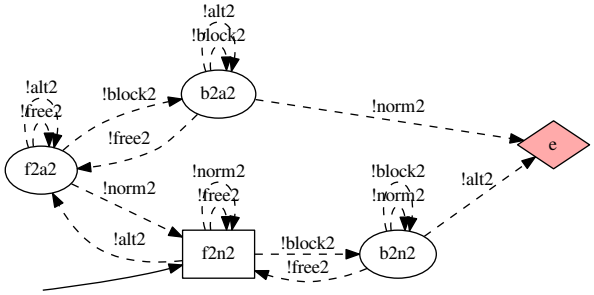


Figure 4.27.: Requirement R_5.

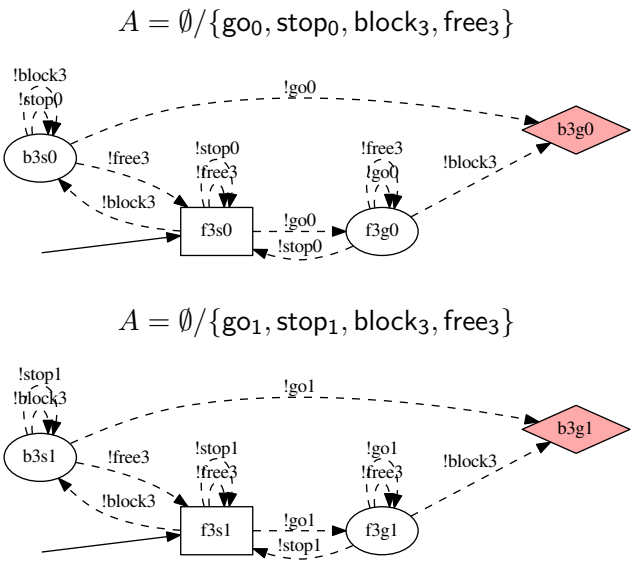


Figure 4.28.: Requirement R_6 results from a conjunction of two EMIAs.

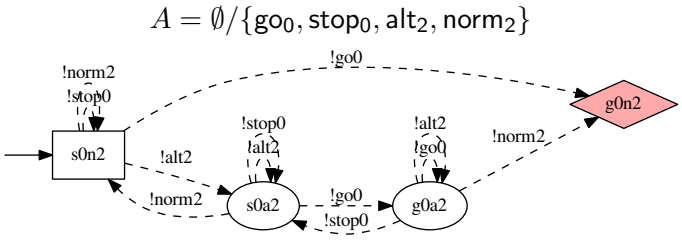


Figure 4.29.: Requirement R_7.

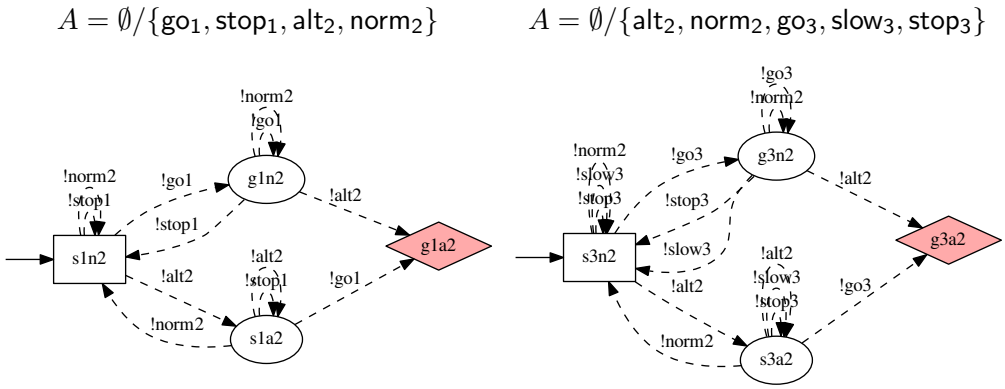


Figure 4.30.: Requirements R_8 (left) and R_9 (right).

$$A = \emptyset/\{\mathsf{go}_0, \mathsf{stop}_0, \mathsf{block}_1, \mathsf{free}_1, \mathsf{go}_1, \mathsf{stop}_1, \mathsf{alt}_2, \mathsf{block}_2, \mathsf{free}_2, \mathsf{norm}_2, \mathsf{block}_3, \mathsf{free}_3, \mathsf{go}_3, \mathsf{slow}_3, \mathsf{stop}_3\}$$

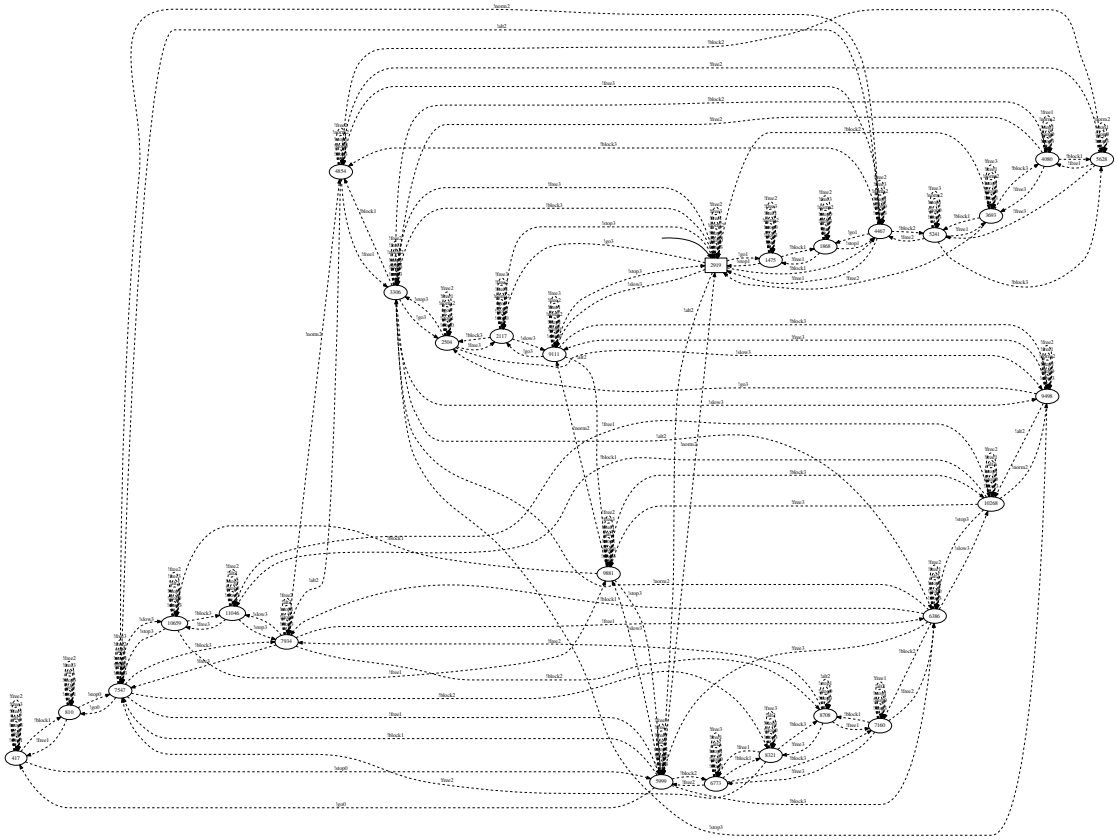


Figure 4.31.: Sketch of the reduced controller specification $R_C(G /\!\!/ S)$; the loops comprise those actions that do not change the state of the related component.

The controller C must satisfy the inequation

$$C \sqsubseteq R_C \sqcap (G /\!\!/ S). \tag{4.9}$$

Concerning the complexity, the conjunctive product underlying R_C has $6 \cdot 8 \cdot 8 \cdot 4^3 \cdot 5 \cdot 4^2 \cdot 4 \cdot 4 \cdot 4 = 6 \cdot 8^2 \cdot 4^8 \cdot 5 = 125\,829\,120$ states and the quotient $G /\!\!/ S$ has 1944 states (as we have seen before). These numbers have to be multiplied in the conjunction of (4.9). From this consideration, the necessity of tool support is obvious, and we have to reduce intermediate results, e.g., by removing unreachable states and merging indistinguishable error states. In order to get an impression, the controller specification we get after applying intermediate reductions is sketched in Figure 4.31, counting 26 states. Remarkably, this final result is relatively small despite the large size of intermediate results. In particular, the whole computation is a question of minutes on a standard laptop hardware (Intel Core i5-4210M, 16 GiB memory) while we were unable to visualise some of the intermediate results within

an arbitrarily chosen time limit of 12 hours. The source code of this case study is found in Appendix C.2.

We summarise the methodology that we employed for designing a railway system. First, we designed railway signals by stepwise refinement, which we composed with switches and blocks to a partial system S. This was possible due to EMIA's support of independent implementability. Second, we employed perspective-based specification in order to combine the requirements R_1 through R_9 via conjunction into an overall safety requirement R. Third, we applied incremental design to extend S by a controller. To do so, we synthesised a controller specification $G /\!\!/ S$ from a global requirement G and the already existing partial system S via quotienting. Finally, we refined the controller specification by conjoining $G /\!\!/ S$ and R yielding a specification C that guarantees the safe operation of the railway system.

4.4. Tool Support

During the work on this thesis, two prototypical tools that implement the Modal Interface Automata have been developed. The main reason, why we developed new tools instead of extending one of the existing tools, is that several major requirements of MIA are insufficiently supported by these tools, e.g., internal transitions, disjunctive must-transitions, nondeterminism and input/output, such that extending these tools would require major refactoring and significantly more effort than a new implementation.

4.4.1. Haskell Implementation of MIA

The MIA theory has been implemented in Haskell [Has17] as a tool called *Gemia*, which is available as free software at [Fen17] and in Appendix B. Gemia has been used to validate the examples in Section 4.2 and to cross-validate the Go implementation presented in Section 4.4.2. The Gemia tool supports the following features:

- Pruning of unreachable and inconsistent states,
- Alphabet extension,
- Parallel product,
- Conjunctive product and conjunction,
- Pre-quotient,
- Visualisation via Graphviz [Gra17].

At the current state of the implementation, the following aspects of MIA are not supported: internal and weak transitions, hiding, restriction, error-pruning and disjunction. In particular, we did not miss error-pruning in practice because we were always interested in the reasons for a communication mismatch.

```
-- Global server specification.
global :: Gemia Int
global = makeGemia (Leaf 0)
  [req 0 Input  "rqst" [1],
   opt 1 Output "fail" 0,
   opt 1 Output "resp" 0]
```

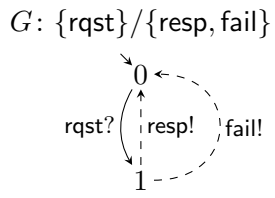


Figure 4.32.: Example of a Gemia specification of the global requirement from Figure 4.1.

Gemia's main data structure used to represent MIAs is based on records and lists. States are represented as binary trees in order to make tracing back to the origin of a state in a composed system easier. An alternative implementation that uses IDs for states and maps instead of lists, as was done in the Go implementation below, did not prove to be a performance benefit.

Gemia is provided as an embedded DSL such that the full expressiveness of the Haskell programming language may be employed for specification. An example of such a specification is given in Figure 4.32 by means of the global requirement of our client-server example from Section 4.2. The specification has the name `global` and is of type `Gemia Int`, i.e., it corresponds to a MIA where the basic type for representing states is `Int`. However, the actual type for node representation is a tree of `Int`, so that the compositional structure resulting from applying MIA-operations is still visible in a composed system. This is the reason why the initial state is given as `Leaf 0`.

The transitions are given as a list of calls to `req` and `opt` for required and optional transitions, respectively. For both functions, the first argument is the source node of the transition, the second is the IO-type, namely `Input` or `Output`, and the third argument is a string that represents the action label of the transition. Target nodes are given as the fourth argument in a slightly different manner for `req` and `opt`. As MIA supports disjunctive must-transitions, `req` expects a list of nodes as a target, while the target for `opt` is just a single node. All nodes, source and target, are given as the basic node type, `Int` in our example. Note that 'required' and 'optional' have a slightly different meaning than 'must' and 'may'. A required transition is a must-transition that automatically includes the may-transitions required by syntactic consistency, while an optional transition represents a may-transition that is not required by any must-transition. This way, it is impossible to specify syntactically inconsistent systems in `Gemia`. The complete source code of the client-server case study is found in Appendix C.1.

4.4.2. Go Implementation of MIA and EMIA

The MIA theory has also been implemented in Go as a part of Gareis' Master's thesis [Gar15], which was supervised by the author. The main goal of this tool, which is called MiaGo, was to extract MIAs from Go source code in order to check them against specifications also given as MIAs. The choice of Go was made due to its direct support of channels similar to those in MIA. This implementation has been extended to EMIA by the author of this dissertation and used to carry out the case study in Section 4.3. It

```
specification Signal {
  states { s, g, u }
  inis { s }
  unis { u }
  errs {   }
  channels { go, stop }
  s {
    go?? -> g
    stop? -> u
  }
  g {
    stop?? -> s
    go? -> u
  }
};
```

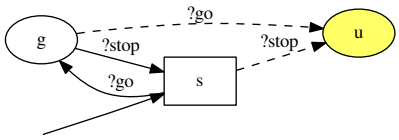


Figure 4.33.: Example of a MiaGo specification of a railway signal from Section 4.3.1.

supports the following operations:

- Pruning of unreachable and inconsistent states,
- Alphabet extension,
- Hiding and restriction,
- Parallel product and parallel composition,
- Conjunctive product and conjunction,
- Pre-quotient and quotient,
- Disjunction,
- Refinement checking via boolean equation systems,
- Construction of counter-examples in case refinement does not hold,
- Extraction of MIAs from Go source code,
- Visualisation via Graphviz [Gra17].

The main data structure employed in MiaGo is very similar to that of the Haskell implementation. States are represented as unique integer IDs and may carry possibly non-unique string identifiers used for displaying.

MiaGo employs an external DSL for specifying interfaces. As an example, Figure 4.33 shows the specification of a signal as designed in our railway case study in Section 4.3.1. The specification is named `Signal`. With the keyword `states` we declare three state identifiers, `s`, `g` and `u`. Only declared state identifiers may be used in the remainder of the specification. The keywords `inis`, `unis` and `errs` are used to declare which states are initial, universal or erroneous, respectively. In our case, no error-state is declared.

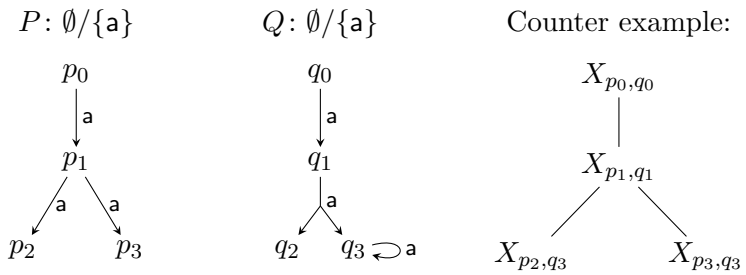


Figure 4.34.: Counter example when checking refinement with boolean equation systems.

With the `channels`-keyword we declare two channel names, `go` and `stop`. Afterwards, the state transitions are defined. For example, state `s` has a `go`-input must-transition (marked with `??`) leading to state `g` and a `stop`-input may-transition (marked with `?`) leading to state `u`. Similarly, output must-transitions are marked with `!!` and output may-transitions with `!`.

Most operators have a straightforward implementation. Therefore, we only describe how refinement checking is implemented. Refinement checking is translated into a *boolean equation system* (BES) [GW05] that is solved with the external solver pbes2bool from the mCRL2 toolset [mCRL17]. A refinement relation $\mathcal{R}$ between two MIAs P, Q by boolean variables $X_{p,q}$, for $p \in S_P$, $q \in S_Q$, where $X_{p,q}$ is true if and only if $(p,q) \in \mathcal{R}$. Based on this representation, the refinement check is encoded as a BES containing, for all $p \in S_P$, $q \in S_Q$, an equation of the form

$$X_{p,q} = \left(\bigwedge_{a \in A} \bigwedge_{\{p' \mid p \xrightarrow{a}_P p'\}} \bigvee_{\{q' \mid q \xrightarrow{a}_Q q'\}} X_{p',q'} \right) \wedge \left(\bigwedge_{a \in A} \bigwedge_{\{Q' \mid q \xrightarrow{a} Q'\}} \bigvee_{\{P' \mid p \xrightarrow{a}_P P'\}} \bigwedge_{p' \in P'} \bigvee_{q' \in Q'} X_{p',q'} \right). \tag{4.10}$$

Hereby, the first conjunct of the equation's right-hand side expresses the refinement conditions on may-transitions and the second conjunct the conditions on must-transitions (see Definition 3.5 on page 37). Further, the equation $X_{p_0,q_0} = \mathsf{true}$ is added in order to include initial states p_0, q_0; otherwise, the empty refinement relation is always a trivial solution of the BES.

The greatest fixed point of such a BES yields the greatest refinement relation between P and Q if the BES has a solution. Otherwise, P does not refine Q and pbes2bool returns a counter-example in the form of a tree of variables $X_{p,q}$. In our case, such a tree is rooted in X_{p_0,q_0} and represents paths that explain why $(p_0, q_0) \notin \mathcal{R}$.

Example 4.1. Consider the MIAs P and Q shown in Figure 4.34. Checking whether P refines Q is translated into the following BES, where we omit equations of uninteresting

node pairs for brevity:

$$
\begin{aligned}
X_{p_0,q_0} &= X_{p_1,q_1} \\
X_{p_1,q_1} &= ((X_{p_2,q_2} \wedge X_{p_2,q_3}) \wedge (X_{p_3,q_2} \wedge X_{p_3,q_3})) \wedge ((X_{p_2,q_2} \wedge X_{p_2,q_3}) \vee (X_{p_3,q_2} \wedge X_{p_3,q_3})) \\
X_{p_2,q_2} &= \mathsf{true} \\
X_{p_3,q_2} &= \mathsf{true} \\
X_{p_2,q_3} &= \mathsf{false} \\
X_{p_3,q_3} &= \mathsf{false}
\end{aligned}
$$

It is easy to see that P does not refine Q. A counter-example tree (out of several possible ones) is also shown in Figure 4.34. Node X_{p_0,q_0} is false due to X_{p_1,q_1} being false because both alternatives X_{p_2,q_3} and X_{p_3,q_3} are false. This counter-example explains that P does not refine Q because neither p_2 nor p_3 refines q_3. Hence, we may focus on a further analysis of states p_2, p_3 and $q3$, revealing that the refinement check fails due the a-loop that is required by q_3 but not implemented by p_2 or p_3.

Further, Gareis [Gar15] extended the Go language by means of a preprocessor that enables one to annotate Go functions with MIAs given in the above DSL. MiaGo is able to extract MIAs from Go source code, where the action labels correspond to the channel names of the program. By checking whether the extracted MIA refines the annotated MIAs one may verify the communication behaviour of a function against a behaviour specification. For further details see [Gar15].

4.4.3. Overview of Existing Tools

The material presented in this section is based on currently unpublished joint work with Gareis [FG17]. We compare different software tools that implement specification theories such as IA, MTS and their extensions. Namely, we consider the MIO Workbench [Bau+10; BML11], Mica [Cai11; Rac+11], Ticc [Adl+06; LAF06], MoTraS [Ben+15; KS13] and MTSA [DIp+07; DIp+08].

The MIO Workbench [Bau+10; BML11] implements Bauer et al.'s MIO as a series of Eclipse plug-ins [Ecl17] in Java. It offers a command line shell as well as a graphical editor provided by an Eclipse perspective. The graphical editor can be used to create and change MIOs, to check for refinement and to investigate errors. The plugin itself offers checking for strong, weak and may-weak modal refinement, optimistic and pessimistic compatibility, and satisfyability. Nondeterminism and the operations operations parallel composition, conjunction and quotient are supported. However, the above research papers leave it unclear how refinement checking and the quotient are implemented. The latest update of the MIO Workbench is from August 2012 and requires Eclipse version 3.7 from 2011.

The tool Mica [Cai11; Rac+11] implements Caillaud et al.'s Modal Interfaces (MI) in OCaml. Interfaces are specified in an embedded DSL such that the full expressiveness of the OCaml language is available for specification. Mica provides a graphical visualisation

of interfaces via Graphviz [Gra17]. Beyond modal refinement and the standard operations of interface theories, subimplication and several temporal operators are also implemented. Many checks like consistency, compatibility, satisfyability, completeness and inconsistency are provided. As the tool is based on MI, it inherits MI's associativity bug and does not support nondeterminism and internal behaviour. The current version is 0.08a, last updated in April 2014.

Ticc [Adl+06; LAF06] implements Sociable Interfaces [Alf+05] in OCaml. A textual language is used to specify interfaces which are stored as MDDs/BDDs employing the CU decision diagram package CUDD [Som15]. The tool implements the operations parallel product, parallel composition and the restriction of input actions. Compatibility of components can be checked. Refinement, modalities, internal behaviour and the operations conjunction, disjunction and quotient are not supported. Unfortunately, the tool is not available anymore. Therefore, our tool description is taken from the previously mentioned citations.

MoTraS [Ben+15; KS13] implements several variants of Larsen et al.'s Modal Transition Systems (MTS), such as disjunctive and parametric MTS, and is written in Java. The tool provides a command line interface to an external DSL and a graphical interface on the NetBeans platform [NB17]. It offers many operators such as conjunction, parallel product, quotient and deterministic hull. Modal refinement checking is translated into quantified boolean formulas (QBF) which are solved with external QBF solvers. LTL model checking and consistency checks are also provided. A unique property of MoTraS is the possibility to handle parametric systems. Internal behaviour is not supported. The current version is 1.0, and its latest update is from September 2014.

MTSA [DIp+08; DIp+07] also realises Larsen et al.'s Modal Transition Systems. It is implemented in Java as an Eclipse plugin [Ecl17]. Eclipse perspectives offer a graphical user interface. In addition, the textual language FSP [MK99] may be used to describe processes and complex systems. LTL model checking is supported, and MTS models may be synthesised from FLTL safety properties [GM03]. The tool supports parallel composition, and refinement checking. Furthermore, it offers checks for deadlocks, safety, and progress. Internal behaviour is not supported. The latest version is from August 2016.

All of the above tools have been implemented for a particular theory and, therefore, differ in many small aspects. Some of them are not maintained anymore, and the last updates have been made years ago. We were unable to install Ticc due to its source code being unavailable, and it was hard to install the MIO Workbench and MoTraS due to the required old versions of software on which the tools depend.

5. Towards a Behavioural Type Theory

Interface theories [AH01a; AH05; Bau+10; Rac+11; Buj+16] provide an incremental, component-based design methodology for the specification of concurrent reactive systems. Being specification-oriented, they employ a uniform interaction model, i.e., one may observe whether an action is performed whereas the internal structure of an action is invisible. When moving towards implementations, one wishes to model interactions that include structured data, e.g., exchanging a typed message or calling a method with typed parameters. Examples of actions in such a structured interaction model are reading and writing typed messages on *communication channels* that are given by *channel names*. Such models have been investigated in various forms of behavioural type systems; an overview can be found in [Hüt+16]. In particular, session types have developed into a promising family of formalisms for providing behavioural type systems for concurrent programming languages [BY09; Car+16; CDY14; CPT16; GH05; Hon93; HYC16; KGG14].

Session types are well-studied behavioural type systems for programming languages based on the π-calculus [MPW92] and support higher order features such as name passing and session delegation. A session type is intended to describe the communication protocol of an interactive session between communicating components, e.g., in service-oriented applications. The major goals of session types are to guarantee the following properties [HYC16]:

- *Session fidelity*: The communication sequence of the interacting components adheres to the communication protocol specified in the session type.
- *Communication safety*: A session is always free of communication errors, i.e., communicating components agree on the type of a message when it is sent and received.
- *Linearity*: A communication channel is always used linearly, i.e., every sent message is received exactly once within a session.
- *Progress*: A session always comes to its intended end, i.e., a session is deadlock-free in the sense that communicating components never get stuck in an intermediate state of the communication protocol.

These guarantees are achieved due to strong assumptions such as a closed systems view via global types, pessimistic compatibility and private communication. Further, session types strictly distinguish branching and selection, which are similar to external and internal choice in CSP [Hoa85]. *Branching* describes several options, e.g., services a

component offers to its communication peers, while *selection* describes a choice that a component may make between the services offered by one of its peers.

In this chapter, we present a first step towards bridging the gap between the more specification-oriented interface theories and the more implementation-oriented session types. We develop de Alfaro and Henzinger's Interface Automata (IA) [AH01a; AH05] into a behavioural type theory similar to session types. In this theory, IA-refinement corresponds to behavioural subtyping and type checking is a computationally efficient approximation of refinement checking. Hence, our behavioural type system is applicable from abstract levels of specification theories down to concrete levels of type theories.

In contrast to session types, we employ an open systems view and optimistic compatibility. This is achieved by integrating the parallel composition operator into the type system deliberating us from the necessity of a manually defined global type. Our type system provides a type-level operational semantics and a more explicit and more precise typing of processes than is usual for session types. Further, we do not separate branching and selection, i.e., any state may specify branching and selection options simultaneously. Instead of private communication, we employ a multicast parallel composition as usual in interface theories [Rac+11; Buj+16]. However, we assume an information flow direction from output to input in contrast to interface theories, which are agnostic with respect to this question (see Remark 5.1). In order to study the differences between interface theories and session types, we present a simple proof of concept that is as close as possible to a typed version of Interface Automata. In particular, we do not consider name passing.

Remark 5.1 (Terminological Pitfall)**.** *The terms input and output have a different meaning in interface theories and session types. In interface theories, an output is active and an input reactive, and—despite the suggestive terms—no information flow direction is considered in these theories due to the uniform communication model. In session types, input, output and selection are active, whereas branching is reactive. In addition, information flow is directed from output to input.*

5.1. The ΠIA Behavioural Type System

In this section we develop the ΠIA behavioural type system, consisting of an implementation language based on the π-calculus and a type language based on IA. A simple way of enriching interface theories with data types is to extend the action alphabet with types. If A is an alphabet of uniform actions and T a set of data types, we may employ the set $A \times T$ as typed action alphabet of an interface. This is possible for any IA-based interface theory. We do so for IA because it is a simple interface theory and sufficient for illustrating the idea.

Definition 5.2 (Interface Automata with Typed Actions)**.** Let A be a set of channel names and T a set of data types. An *Interface Automaton with Typed Actions* (IATA) is an interface automaton (seen as a subclass of EMIA) with alphabets $I \cup O \subseteq A \times T$. We write $a : t$ for $(a, t) \in A \times T$, $a?t$ for $(a, t) \in I$, $a!t$ for $(a, t) \in O$, and call such a tuple a *typed action*.

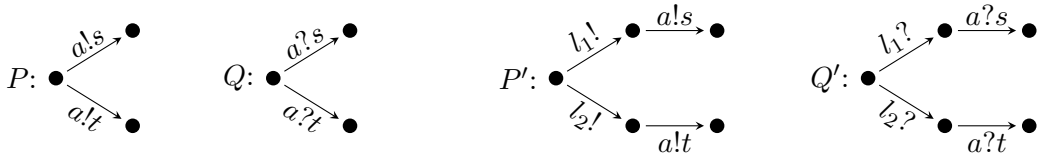


Figure 5.1.: Should nondeterministic specifications be considered compatible?

While nondeterminism is quite natural for IA, we face some subtleties if the actions are typed. For example, it is debatable whether the behavioural types P and Q in Figure 5.1 should be considered compatible. Considering P and Q compatible presupposes some kind of type reflection because Q must know whether the object that P is sending over channel a is of type s or t in order to choose the appropriate transition. Hence, Q's choice is determined by P and cannot be considered nondeterministic anymore. This consideration also applies to IATA because $a!s$ and $a!t$ are different actions if s and t are different types.

In session types this issue is solved by employing a strict way of determinising such nondeterministic choices. Transitions are uniquely labelled on the implementation *and* the type level [HYC16]. Such a labelling is made explicit in an IA-based setting as shown in specifications P' and Q' of Figure 5.1. For these reasons, we restrict our investigation to deterministic systems in this chapter, which is necessary in the proof of Proposition 5.14. For the same reason we exclude internal transitions labelled τ.

5.1.1. The Type Language

While IATAs may serve as a type language for typing processes, they are not optimal for type checking because they deviate significantly from the syntactic structure of an implementation language suitable for programming. For example, the behaviour of a channel is modelled only implicitly in IATA. Therefore, we first define an alternative type language and investigate its relation to IATAs in Section 5.1.5.

Definition 5.3 (ΠA Type Language)**.** The ΠA type language is defined by the following grammar:

$$
\begin{array}{lcll}
\mathsf{T} & ::= & \mathsf{D} \mid \mathsf{C} \mid \mathsf{P} & \text{(general types)} \\
\mathsf{D} & ::= & t_1 \mid t_2 \mid \ldots & \text{(data types)} \\
\mathsf{C} & ::= & \mathsf{end} \mid ?\bar{\mathsf{T}}.\,\mathsf{C} \mid !\bar{\mathsf{T}}.\,\mathsf{C} \mid \mathsf{C}+\mathsf{C} \mid & \\
 & & \mathsf{C} \parallel \mathsf{C} \mid \alpha \mid \mu\alpha.\,\mathsf{C} & \text{(channel types)} \\
\mathsf{P} & ::= & \mathsf{S} \mid \mathsf{P} \parallel \mathsf{P} & \text{(process types)} \\
\mathsf{S} & ::= & \mathsf{err} \mid \alpha \mid \mu\alpha.\,\mathsf{A} \mid \mu\alpha.\,\mathsf{err} \mid \mathsf{A} & \text{(sequential types)} \\
\mathsf{A} & ::= & \mathsf{end} \mid \mathsf{N}?\bar{\mathsf{T}}.\,\mathsf{S} \mid \mathsf{N}!\bar{\mathsf{T}}.\,\mathsf{S} \mid \mathsf{A}+\mathsf{A} & \text{(interaction types)} \\
\mathsf{N} & ::= & n_1 \mid n_2 \mid \ldots & \text{(channel names)}
\end{array}
$$

In this thesis, $\bar{\mathsf{T}} = \mathsf{D}$; however, we envision $\bar{\mathsf{T}} = \mathsf{T}$ for future work. We assume that the

precedence of the operators decreases from left to right as follows: $. > \| > + > \mu$.

A ΠA type T is either a data type, a channel type or a process type. Data types are the types of a standard type language such as a typed λ-calculus. A channel type describes how a channel may be used: type end denotes a terminated channel; $?\bar{\mathsf{T}}.\,\mathsf{C}$ means receiving an action of type form $\bar{\mathsf{T}}$ on the channel and then using the channel like C; a type of form $!\bar{\mathsf{T}}.\,\mathsf{C}$ means performing an action observable as type form $\bar{\mathsf{T}}$ on the channel and then using the channel like C; types of forms $\mathsf{C}+\mathsf{C}$ and $\mathsf{C}\|\mathsf{C}$ denote branching and parallel composition, respectively; α is a channel variable and $\mu\alpha.\,\mathsf{C}$ describes the recursive usage of a channel. A process type is either a sequential (process) type or a parallel process type. Sequential types are structured as follows: err denotes an error type; α is a process variable and $\mu\alpha.\,\mathsf{S}$ denotes a recursive process type. Interaction types describe how processes may interact with their environment: end means termination; a type of form $\mathsf{N}?\bar{\mathsf{T}}.\,S$ denotes receiving an action of type $\bar{\mathsf{T}}$ on channel N and then behaving as S; type form $\mathsf{N}!\bar{\mathsf{T}}.\,\mathsf{S}$ means acting as $\bar{\mathsf{T}}$ on channel N and then behaving as S; type $\mathsf{A}+\mathsf{A}$ means branching. Finally, N are channel names. We do not distinguish the channel-level from the process-level symbols end, $\|$, $+$, $?$, $!$, α and μ because they are easily disambiguated from the context. Also, we often omit the symbol end, e.g., we write $\mathsf{N}!\bar{\mathsf{T}}$ instead of $\mathsf{N}!\bar{\mathsf{T}}.\,\mathsf{end}$. We also use different letters than $t_1, t_2, \ldots$ or $n_1, n_2, \ldots$ for data types and channel names, which will always be clear from the context.

Some syntactically different type expressions may semantically denote the same type. Hence, we need equivalence rules that abstract from such purely syntactic differences. To do so, we define the concept of free variables:

Definition 5.4 (Free Variables)**.** The set of *free variables* of a type expression is defined recursively by the following rules:

$$\begin{aligned} \mathrm{fv}(\mathsf{end}) = \mathrm{fv}(\mathsf{err}) = \emptyset, && \mathrm{fv}(x) = \{x\}, \\ \mathrm{fv}(n!y.\,S) = \mathrm{fv}(S), && \mathrm{fv}(n?y.\,S) = \mathrm{fv}(S) \setminus \{y\}, \\ \mathrm{fv}(S \,\|\, T) = \mathrm{fv}(S + T) = \mathrm{fv}(S) \cup \mathrm{fv}(T), && \mathrm{fv}(\mu x.\,S) = \mathrm{fv}(S) \setminus \{x\}. \end{aligned}$$

Now we can define structural equivalence for types:

Definition 5.5 (Type-level Structural Equivalence)**.** Let S, T and U be type expressions according to Definition 5.3. Two types are *structurally equivalent* if they are equivalent according to the following rules:

$$\begin{aligned} S + T &\equiv T + S & (S + T) + U &\equiv S + (T + U) \\ S \,\|\, T &\equiv T \,\|\, S & (S \,\|\, T) \,\|\, U &\equiv S \,\|\, (T \,\|\, U) \\ S + \mathsf{end} &\equiv S & S &\equiv S \\ S \,\|\, \mathsf{end} &\equiv S & S \,\|\, \mathsf{err} &\equiv \mathsf{err} \\ \mu x.\,\mathsf{end} &\equiv \mathsf{end} & \mu x.\,\mathsf{err} &\equiv \mathsf{err} \\ \mu x.\,S &\equiv S[\mu x.\,S/x] & \mu x.\,S \,\|\, T &\equiv S \,\|\, \mu x.\,T, \text{ if } x \notin \mathrm{fv}(S) \end{aligned}$$

For traditional data type systems, the subject reduction property means that the type of an expression stays stable under evaluation of the expression. For example, evaluating expression '$3 + 4$' of type integer yields expression '7', which still is of type integer. A common application of subject reduction is that one does not need to evaluate an expression operationally in order to compute its type. In contrast, a behavioural type captures the dynamics of an object and, thus, may change when the object is used. For behavioural types, the stability requirement of the subject reduction property means that such changes may only happen in a controlled way, i.e., we need an operational semantics on the type level that conforms with the implementation-level operational semantics. In addition, the usual interface-theoretic synchronisation of input and output happens at this operational semantics level.

Definition 5.6 (Type-level Operational Semantics)**.** The operational semantics $[\![S]\!]$ of a ΠIA behavioural type S is an LTS defined by the following rules:

$$(n!s.\, t) + r \xrightarrow{n!s} t \tag{5.1}$$

$$(n?s.\, t) + r \xrightarrow{n?s} t \tag{5.2}$$

$$((n!s.\, t) + t') \parallel (n?s.\, r + r') \xrightarrow{n!s} t \parallel r \tag{5.3}$$

$$((n?s.\, t) + t') \parallel (n?s.\, r + r') \xrightarrow{n?s} t \parallel r \tag{5.4}$$

It is obvious that the operational semantics of a behavioural type defines an IATA. Hence, we can lift the interface-theoretic compatibility to types:

Definition 5.7 (Compatibility of Types)**.** A process type T is *optimistically compatible* if its operational semantics $[\![T]\!]$ is optimistically compatible, i.e., if errors may not be reached by only output transitions.

5.1.2. The Implementation Language

Similar to session types, we base our implementation language on a variant of the π-calculus.

Definition 5.8 (ΠIA Implementation Language)**.**

$$\begin{array}{lcll} \mathsf{P} & ::= & \mathsf{S} \mid \mathsf{P} \parallel \mathsf{P} & \text{(general processes)} \\ \mathsf{S} & ::= & \mathsf{A} \mid x \mid \mu x.\, \mathsf{P} \mid \mathsf{err} & \text{(sequential processes)} \\ \mathsf{A} & ::= & \mathsf{end} \mid \mathsf{N}?y.\mathsf{S} \mid \mathsf{N}!y.\mathsf{S} \mid \mathsf{A} + \mathsf{A} & \text{(process interactions)} \\ \mathsf{N} & ::= & n_1 \mid n_2 \mid \ldots & \text{(channel names)} \end{array}$$

Analogous to the type language, we have parallel composition, recursion, error, termination, observation, action, and branching. Further, the implementation language shares the *same* channel names with the type language. Note that, in contrast to Definition 5.3, an explicit $\mu x.\, \mathsf{err}$ is not required in Definition 5.8 because this expression is already

included in $\mu x.\,\mathsf{P}$. The implementation-level structural equivalence follows the same rules as type-level structural equivalence (see Definition 5.5), except that S, T and U denote processes instead of types, and we denote it by the same symbol $\equiv$. The operational semantics is then defined as follows:

Definition 5.9 (Implementation-level Operational Semantics)**.** Given variables y and z, where we assume w.l.o.g. that $y \notin \mathrm{fv}(Q)$, the implementation-level operational semantics is defined by the following rules:

$$(n!y.\,P) + Q \xrightarrow{n!y} P \tag{5.5}$$

$$(n?y.\,P) + Q \xrightarrow{n?y} P \tag{5.6}$$

$$(n!y.\,P + P') \parallel (n?z.\,Q + Q') \xrightarrow{n!y} P \parallel Q[y/z] \tag{5.7}$$

$$(n?y.\,P + P') \parallel (n?z.\,Q + Q') \xrightarrow{n?y} P \parallel Q[y/z] \tag{5.8}$$

Because we take an open systems view as is usual in interface theories, the operational semantics rules 5.5 and 5.6 capture actions which are not synchronised. In (5.8) we need the substitution $[y/z]$ in order to synchronise the information received from the environment of $P \parallel Q$, which may also communicate on n.

5.1.3. The Type System

The purpose of the type language is to constrain the validity of expressions of the implementation language to *well-typed* or *typable* expressions. This is achieved via typing rules that relate valid expressions to types. We employ typing contexts in order to reason about the type of free variables and channel names:

Definition 5.10 (Typing Context)**.** Given variables or channel names $x_1, \ldots, x_k$ and type expressions $T_1, \ldots, T_k$, a *typing context* is a collection $\Gamma := x_1 : T_1, \ldots, x_k : T_k$, where $x : T$ means that variable x has type T.

We do not distinguish typing contexts that differ only with respect to ordering or multiplicity of elements. In particular, our type system is not in Curry-Howard correspondence with a purely linear logic, which would constrain us to private communication as in session types [CPT16; Car+16]. We also need operations for merging contexts:

Definition 5.11 (Merging of Contexts)**.** For typing contexts $\Gamma = x_1 : S_1, \ldots, x_k : S_k$ and $\Gamma' = y_1 : T_1, \ldots, y_l : T_l$, we define $\Gamma + \Gamma'$ and $\Gamma \parallel \Gamma'$ as the smallest contexts satisfying the following rules:

CB1. if $z : S \in \Gamma$ and $z \notin \Gamma'$, then $z : S \in \Gamma + \Gamma'$,
CB2. if $z : T \in \Gamma'$ and $z \notin \Gamma$, then $z : T \in \Gamma + \Gamma'$,
CB3. if $z : S \in \Gamma$ and $z : T \in \Gamma'$, then $z : S + T \in \Gamma + \Gamma'$.

CP1. if $z : S \in \Gamma$ and $z \notin \Gamma'$, then $z : S \in \Gamma \parallel \Gamma'$,

$$(\text{ax})\frac{}{x : S \vdash x : S} \qquad (\text{weak})\frac{\Gamma, \Delta \vdash x : S}{\Gamma, y : T, \Delta \vdash x : S} \quad [y \notin \text{fv}(x)]$$

$$(\text{end})\frac{}{\Gamma \vdash \mathsf{end} : \mathsf{end}} \qquad (\text{err})\frac{}{\Gamma \vdash \mathsf{err} : \mathsf{err}}$$

$$(\text{out})\frac{\Gamma \vdash y : S \qquad \Gamma, x : T \vdash P : R}{\Gamma, x : !S.\, T \vdash (x!y.\, P) : x!S.\, R} \qquad (\text{in})\frac{\Gamma, x : T, y : S \vdash P : R}{\Gamma, x : ?S.\, T \vdash (x?y : S.\, P) : x?S.\, R}$$

$$(\text{bra})\frac{\Gamma \vdash P : S \qquad \Gamma' \vdash Q : T}{\Gamma + \Gamma' \vdash (P + Q) : S + T} \qquad (\text{par})\frac{\Gamma \vdash P : S \qquad \Gamma' \vdash Q : T}{\Gamma \parallel \Gamma' \vdash (P \parallel Q) : S \parallel T}$$

$$(\text{rec})\frac{\Gamma, x : \alpha \vdash E : T}{\Gamma \vdash (\mu x.\, E) : \mu\alpha.\, T}$$

Figure 5.2.: The IIIA typing rules.

CP2. if $z : S \in \Gamma'$ and $z \notin \Gamma$, then $z : S \in \Gamma \parallel \Gamma'$,
CP3. if $z : S \in \Gamma$ and $z : T \in \Gamma'$, then $z : S \parallel T \in \Gamma \parallel \Gamma'$.

Note that $S + T$ is only defined if S and T are types of a kind that may be composed according to Definition 5.3. Otherwise, $\Gamma + \Gamma'$ is not defined and the expression cannot be typed. The same is true for $\parallel$. The merging operations allow us to define the IIIA typing rules:

Definition 5.12 (Typing Rules)**.** An expression E of the implementation language is *well-typed* if and only if there is a context Γ and a type expression T such that $\Gamma \vdash E : T$ is derivable by the rules shown in Figure 5.2.

The structural rules (ax) and (weak) are standard. Rules (end) and (err) ensure that, in any context, a terminated process is of type end and an error process is of type err. Rule (out) is concerned with outputting a variable $y : S$ on a channel x. Note how the behavioural type of x changes when applying the rule. Rule (in) is the corresponding input rule which mainly differs from (out) by the fact that variable y is bound by the application of (in). Branching (bra) and parallel composition (par) are straightforward. In particular, (par) does not require a compatibility check due to our explicit error representation. Rule (rec) is a standard recursion rule, where E is either a process or a channel name.

Example 5.13. We consider a client-server application where a client may send an account-id i via a channel q (query) to a server that responds with an account-balance b via a channel r (response). A possible client implementation is $C = q!i.\, r?y.\, \mathsf{end}$ and a possible server implementation $S = q?j.\, r!b.\, \mathsf{end}$. We show that the composition $C \parallel S$ is

well-typed. The typing of the client due to rules (ax), (in) and (out) is:

$$\dfrac{\dfrac{}{i:\mathsf{string}, r:?\mathsf{int}.\,\mathsf{end} \vdash i:\mathsf{string}} \quad \dfrac{\dfrac{}{q:\mathsf{end}, i:\mathsf{string}, r:\mathsf{end}, y:\mathsf{int} \vdash \mathsf{end}:\mathsf{end}}}{q:\mathsf{end}, i:\mathsf{string}, r:?\mathsf{int}.\,\mathsf{end} \vdash (r?y:\mathsf{int}.\,\mathsf{end}) : r?\mathsf{int}.\,\mathsf{end}}}{q:!\mathsf{string}.\,\mathsf{end}, i:\mathsf{string}, r:?\mathsf{int}.\,\mathsf{end} \vdash (q!i.\,r?y:\mathsf{int}.\,\mathsf{end}) : q!\mathsf{string}.\,r?\mathsf{int}.\,\mathsf{end}.}$$

We abbreviate the last sequent with $\Gamma \vdash C : T_C$. A similar derivation yields the following typing for the server, which we abbreviate with $\Delta \vdash S : T_S$:

$$q:?\mathsf{string}.\,\mathsf{end}, b:\mathsf{int}, r:!\mathsf{int}.\,\mathsf{end} \vdash (q?j:\mathsf{string}.\,r!b.\,\mathsf{end}) : q?\mathsf{string}.\,r!\mathsf{int}.\,\mathsf{end}.$$

Applying rule (par) yields $\Gamma \parallel \Delta \vdash C \parallel S : T_C \parallel T_S$.

5.1.4. Subject Reduction, Congruence and Type Safety

Subject reduction and congruence are two important properties of a type system in that they guarantee that the type language and the implementation language fit well together.

Subject reduction ensures stability of the type under evaluation of an object. For behavioural types, this means that an implementation may only engage in behaviour that is permitted by its behavioural type.

Proposition 5.14 (Subject Reduction)**.** *Let* $\dagger \in \{!, ?\}$, $\Gamma \vdash P : S$ *and* $y : T$. *Then,* $P \xrightarrow{n\dagger y} P'$ *implies the existence of a context* Γ' *and a type* S' *such that* $S \xrightarrow{n\dagger T} S'$ *and* $\Gamma' \vdash P' : S'$.

Proof. Assume $\Gamma \vdash P : S$ and $y : T$. A transition $P \xrightarrow{n\dagger y} P'$ is due to one of the operational semantics rules (5.5)–(5.8):

Rule (5.5): $P = n!y.\,P' + Q$. Applying (bra) yields $\Gamma = \Gamma_1 + \Gamma_2$ and $S = S_1 + S_2$ with $\Gamma_1 \vdash n!y.\,P' : S_1$ and $\Gamma_2 \vdash Q : S_2$. By rule (out), we get $\Gamma_1 = \tilde{\Gamma}, n : !T.\,T'$ and $S_1 = n!T.S'$ with premises $\Gamma' := \tilde{\Gamma}, n : T' \vdash P' : S'$ and $\tilde{\Gamma} \vdash y : T$. By rule (5.1) we establish $S \xrightarrow{n!T} S'$.

Rule (5.6): $P = n?y.\,P' + Q$. By applying typing rules (bra) and (in), we establish $\Gamma = \tilde{\Gamma}, n : ?T.\,T', y : T$ and $S = n?T.\,S' + R$ as well as $\tilde{\Gamma}, n : T', y : T \vdash P' : S'$. Rule (5.2) implies $S \xrightarrow{n?T} S'$.

Rule (5.7): $P = (n!y.\,R_1 + R_2) \parallel (n?z.\,Q_1 + Q_2)$ and $P \xrightarrow{n!y} R_1 \parallel Q_1[y/z]$. A derivation of $\Gamma \vdash P : S$ is due to typing rule (par) with $\Gamma = \tilde{\Gamma} \parallel \tilde{\Delta}$, where $\tilde{\Gamma}$ is obtained by (out) and (bra) from the derivation

$$\dfrac{\dfrac{\Gamma_1 \vdash y : T \qquad \Gamma_1, n : T' \vdash R_1 : U_1}{\Gamma_1, n : !T.\,T' \vdash n!y.\,R_1 : n!T.\,U_1} \qquad \Gamma_2 \vdash R_2 : U_2}{\underbrace{\Gamma_1, n : !T.\,T' + \Gamma_2}_{\tilde{\Gamma}} \vdash n!y.\,R_1 + R_2 : n!T.\,U_1 + U_2}$$

and $\tilde{\Delta}$ from a similar derivation of $\Delta_1, n : ?T.\,T' + \Delta_2 \vdash n?z.\,Q_1 + Q_2 : n?T.\,S_1 + S_2$ due to (in) and (bra). We choose $\Gamma' = (\Gamma_1, n : T') \parallel (\Delta_1, n : T', z : T)$.

From the above derivations we conclude that $S = V \parallel W$ for types $V = n!T.U_1 + U_2$ and $W = n?T.\, S_1 + S_2$. Note that y and z are of the same type because V and W are deterministic and synchronise on n. Hence, (5.3) applies and $S \xrightarrow{n!T} S' := U_1 \parallel S_1$. Rule (par) implies $\Gamma' \vdash R_1 \parallel Q_1[y/z] : S'$.

Rule (5.8): similar to (5.7) but easier. □

Congruence relates the type-level and the implementation-level structural equivalence:

Proposition 5.15 (Congruence)**.** *If $P \equiv Q$ and $\Gamma \vdash P : S$, then there is a $T \equiv S$ such that $\Gamma \vdash Q : T$.*

Proof. $P + Q \equiv Q + P$: obvious due to typing rule (bra).
$(P + Q) + R \equiv P + (Q + R)$: easy due to structural equivalence on types.
$P \parallel Q \equiv Q \parallel P$: obvious due to typing rule (par).
$(P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R)$: easy due to structural equivalence on types.
$P \parallel \mathsf{end} \equiv P$: easy due to structural equivalence on types.
$P \parallel \mathsf{err} \equiv \mathsf{err}$: easy due to structural equivalence on types.
$\mu x.P \equiv P[\mu x.P/x]$: easy due to structural equivalence on types. □

The purpose of a type system is to ensure type safety, in the sense that a typable program cannot go wrong, for some definition of 'wrong'. Our definition of 'wrong' is concerned with the reachability of errors in the operational semantics:

Proposition 5.16 (Type Safety)**.** *A ΠA program that is typable with an optimistically compatible type never reduces to an error autonomously, i.e., by taking only output transitions.*

Proof. By Proposition 5.14, a typable program $P : t$ reduces to err if and only if its type t reduces to err. By optimistic compatibility of t, such errors are guarded by input actions. Hence, P does not reduce to err autonomously. □

A direct consequence of this proof is that closed systems are error-free, where we consider a system as closed if all inputs are bound, i.e., only output actions remain:

Corollary 5.17. *A closed program typable with an optimistically compatible type is error free.*

It is important to include optimistic compatibility in our understanding of type safety. Omitting this requirement would adhere to pessimistic compatibility as is done in session types, where it is sufficient because only closed systems are considered. In contrast, our goal is to develop a behavioural type theory that—beyond ensuring type safety—is also suitable as a compositional design tool. This goal is supported by employing a compositional open systems view. In this view, optimistic compatibility allows one to type intermediate results, i.e., subterms of a term, which may or may not result in an error depending on the environment.

5.1.5. Subtyping and Relation to Interface Automata

In general, subtyping on data types is not compositional for behavioural types. For example, consider a data type Num that has subtypes Int and Float. If one regards processes $p_0 \xrightarrow{a!\mathsf{Num}} p_1$ and $q_0 \xrightarrow{a?\mathsf{Num}} q_1$ compatible, their composition is $p_0 \parallel q_0 \xrightarrow{a!\mathsf{Num}} p_1 \parallel q_1$. However, the refinements $p_0' \xrightarrow{a!\mathsf{Int}} p_1'$ and $q_0' \xrightarrow{a?\mathsf{Float}} q_1'$ of p_0 and q_0 cannot be composed. An approach that is contravariant in either input or output does not solve this problem neither, because it violates the basic idea of becoming more precise when refining. Therefore, we only allow behavioural subtyping. Note, however, that one could relax this restriction to certain kinds of quantitative data types, such as intervals, as described in [Fah+15].

Definition 5.18 (Behavioural Subtyping)**.** The *subtype relation* $\preceq$ between behavioural types is defined by the following rules:

S1. if $S \equiv T$, then $S \preceq T$,
S2. $S \preceq S + n!T.\, T'$,
S3. $S + n?T.\, T' \preceq S$,
S4. if $S \preceq S'$ and $T \preceq T'$, then $S \parallel T \preceq S' \parallel T'$,
S5. if $S \preceq T$, then $n\dagger R.\, S \preceq n\dagger R.\, T$, for $\dagger \in \{?, !\}$,
S6. if $S \preceq T$, then $\mu x.\, S \preceq \mu x.\, T$.

In case of contexts $\Gamma = x_1 : S_1, \ldots, x_k : S_k$ and $\Delta = x_1 : T_1, \ldots, x_k : T_k$ we write $\Gamma \preceq \Delta$ if $S_1 \preceq T_1, \ldots, S_k \preceq T_k$.

Example 5.19. We give an example of subtyping by means of a server S' that, beyond the service provided by the server S of type T_S from Example 5.13, provides an additional service s for sending an amount m of money to the account. The sequent $\Delta' \vdash S' : T'$ derives the typing of S' with the context $\Delta' := q : ?\mathsf{string}.\,\mathsf{end}, b : \mathsf{int}, r : !\mathsf{int}.\,\mathsf{end}, s : ?\mathsf{string}.\,?\mathsf{int}.\,\mathsf{end}$, the implementation $S' := (q?j : \mathsf{string}.\, r!b.\,\mathsf{end}) + (s?j : \mathsf{string}.\, s?m : \mathsf{int}.\,\mathsf{end})$ and the type $T' := (q?\mathsf{string}.\, r!\mathsf{int}.\,\mathsf{end}) + (s?\mathsf{string}.\, s?\mathsf{int}.\,\mathsf{end})$. Obviously, we have $T' \preceq T_S$ by Rule S3.

Axiom S4 requires compositionality of subtyping with respect to parallel composition directly because parallel composition is not defined operationally here. However, we have to show that the syntactic definition of parallel composition corresponds to the operational one:

Proposition 5.20. *Parallel composition is homomorphic with respect to operational semantics, i.e.,* $[\![S \parallel T]\!] = [\![S]\!] \otimes [\![T]\!]$.

Proof. A transition in $[\![S \parallel T]\!]$ is due to one of the rules (5.3) or (5.4). The proofs are similar in both cases, hence, we only show the case of rule (5.3). By assumption, $S = (n!R.\, S_1) + S_2$ and $T = (n?R.\, T_1) + T_2$ and, by (5.1) and (5.2), there are transitions $S \xrightarrow{n!R} S_1$ in $[\![S]\!]$ and $T \xrightarrow{n?R} T_1$ in $[\![T]\!]$, respectively. Due to the definition of the IA parallel product, there is a transition $S \parallel T \xrightarrow{n!R} S_1 \parallel T_1$ in $[\![S]\!] \otimes [\![T]\!]$.

A transition in $[\![S]\!] \otimes [\![T]\!]$ is due to one of the composition rules of IA. Again, we discuss only the most interesting case because the proofs of the other cases proceed analogously. A transition $S \parallel T \xrightarrow{n!R} S_1 \parallel T_1$ is, without loss of generality, due to transitions $S \xrightarrow{n!R} S_1$ and $T \xrightarrow{n!R} T_1$. Depending on the structure of S and T, one of the rules (5.3) or (5.4) applies to $S \parallel T$. Assuming it is rule (5.3), this yields a transition $S \parallel T \xrightarrow{n!R} S_1 \parallel T_1$ in $[\![S \parallel T]\!]$. The other rule follows the same line of argument. □

We may also compare ΠIA types with respect to IATA refinement: Propositions 5.21 and 5.22 relate structural equivalence and subtyping to refinement.

Proposition 5.21 (Structural Equivalence and Refinement)**.** *If $S \equiv T$, then $[\![S]\!] \equiv [\![T]\!]$.*

Proof. By Definitions 5.5 and 5.6, $S \equiv T$ implies $[\![S]\!] = [\![T]\!]$. Thus, $[\![S]\!] \equiv [\![T]\!]$ □

Hence, structural equivalence is finer than mutual refinement.

Proposition 5.22 (Subtyping and Refinement)**.** *If S and T are ΠIA types with operational semantics $[\![S]\!]$ and $[\![T]\!]$, respectively, then $S \preceq T$ implies $[\![S]\!] \sqsubseteq [\![T]\!]$.*

Proof. Straightforward application of the rules of Definition 5.18. □

The subtyping preorder is finer than the refinement preorder, e.g., subtyping distinguishes the types $\mu x.\, !a.\, x$ and $\mu x.\, !a.\, !a.\, x$, which are equivalent under refinement. That is, subtyping yields a syntactic approximation of semantic equivalence. If one is willing to sacrifice efficiency of the subtype check, subtyping can also be replaced by refinement. As a consequence, our behavioural type theory inherits all refinement-related operations of interface theories, such as conjunction and quotienting. Further, this result motivates the following definition:

Definition 5.23 (Implementation Relation)**.** A process P *implements* a type T, written $P \vDash T$, if and only if there are a context Γ and a type S such that $\Gamma \vdash P : S$ and $S \preceq T$.

Obviously, if $\Gamma \vdash P : S$, then $P \vDash S$. Further, an implementation of a type S implements any of S's supertypes:

Proposition 5.24 (Safe Substitution)**.** *If $P \vDash S$ and $S \preceq T$, then $P \vDash T$.*

Proof. By Definition 5.23, $P \vDash S$ implies a derivation $\Gamma \vdash P : S'$ for a type $S' \preceq S$. By transitivity of $\preceq$ and $S \preceq T$, we have $S' \preceq T$. Hence, $P \vDash T$. □

For instance, the server implementation $S' : T'$ of example 5.19 implements the type T_S of the server S from example 5.13 because $T' \preceq T_S$. The additional service provided by S' is harmless, because any client that safely interoperates with S also does so with S'.

As a direct consequence of Propositions 5.22 and 5.24, we get a semantic version of Proposition 5.24:

Corollary 5.25 (Design and Implementation)**.** *If $P \vDash S$ and $[\![S]\!] \sqsubseteq [\![T]\!]$, then there is a type $S_P \preceq S$ such that $P : S_P$ and $[\![S_P]\!] \sqsubseteq [\![T]\!]$.*

This is an important result relating design and implementation: One may consider $[\![T]\!]$ as a representation of a global requirement that has been refined to $[\![S]\!]$ during design. An implementation P of $[\![S]\!]$ represented by its type $S_P \preceq S$ also refines $[\![T]\!]$ and, therefore, satisfies the global requirement. In particular, Corollary 5.25 guarantees the following: if $[\![S_1 \parallel S_2]\!] \sqsubseteq [\![T]\!]$, $P_1 \vDash S_1$ and $P_2 \vDash S_2$, then there is a type $S_{P_1 \parallel P_2}$ with $[\![S_{P_1 \parallel P_2}]\!] \sqsubseteq [\![T]\!]$. Note that the difference between $:$ and $\vDash$ matters here, because $P : S_P$ may be derived with typing rules, while deciding $P \vDash S$ is a more complex operation.

5.2. Discussion and Related Work

In this section we discuss progress, which is a central property of session types, in the light of subtyping and refinement. We also revisit error abstraction in the context of type safety and compare ΠIA with Go as implementation languages. Finally, we discuss related work.

5.2.1. Progress: From Interface Automata to Modal Interface Automata

As a proof-of-concept, we restricted ourselves to Interface Automata instead of Modal Interface Automata because we think that the many details of modal interface theories would distract too much from our main concern of behavioural type theories. Due to the lack of output must-transitions, ΠIA suffers the same weaknesses as IA with respect to progress:

Proposition 5.26 (Progress)**.** *If $\Gamma \vdash P : S$ and $S \xrightarrow{n!T} S'$, then there is a context Γ', a process P' and a variable $x : T$ such that $P \xrightarrow{n!x} P'$ and $\Gamma' \vdash P' : S'$.*

Proof sketch. Similar to the proof of Proposition 5.14, but in the reverse direction, i.e., from type-level transitions to implementation-level transitions. □

Although one can prove Proposition 5.26, it does not generalise to subtyping, i.e., the has-type relation $:$ cannot be replaced by $\vDash$. This is because output transitions may be removed when subtyping. Analogous to Interface Automata, every behavioural type has a trivial black hole subtype where all outputs have been removed [AH01a]. An implementation of such a subtype will never make any progress. This issue also affects session types, which—to the best of our knowledge—is not discussed in the session type literature.

Due to the distinction between may- and must-transitions, a behavioural type theory based on Modal Interface Automata may guarantee progress with respect the must-transitions of the behavioural type.

5.2.2. Error Abstraction

Applying our results of Section 3.2 on error abstraction, we may include the pruning operation directly in our type system by propagating errors backwards. Hence, if S and

T are incompatible, then $S \parallel T = \mathsf{err}$. Type safety (see Proposition 5.16) then gets the following appealing form:

Proposition 5.27 (Type Safety II)**.** *If $P : T$ and $T \neq \mathsf{err}$, then P does not reduce to an error.*

Proof. Analogous to the proof of Proposition 5.16. □

This version corresponds to the original way of defining parallel composition in IA. However, the considerations with respect to redefining errors to non-erroneous behaviour discussed in Section 3.2 also apply here.

5.2.3. Go as Implementation Language

We implemented MIA in the Go programming language [Go17]. The choice of Go was due to its first order support of channels with synchronous communication similar to that of MIA. However, when comparing Go and ΠIA as implementation languages, there are still significant differences that are relevant to a formal type system.

The Go equivalent of a process is called a goroutine. Goroutines are a kind of lightweight threads permitting the parallel execution of Go functions. Goroutines may communicate via FIFO channels with a buffer size that is usually fixed at channel creation. If the buffer size of a channel is zero, the channel is unbuffered. In contrast to a process in interface theories, where the input and output alphabets are disjoint, a goroutine is not required to use different channels for input and output. However, Go does not support multicast or broadcast communication.

Synchronous communication between goroutines may be achieved by using unbuffered channels. A goroutine that makes an output on an unbuffered channel blocks until some other goroutine reads from the channel. Vice versa, if one goroutine tries to input from an unbuffered channel, it blocks until a different goroutine makes an output to the channel. It is possible to wait for input and output on several channels simultaneously by means of the select statement. A communication mismatch in the sense of interface theories corresponds to a blocked output process. However, because the environment may still make progress, the blocked process may be unblocked if the environment reads from that channel in some subsequent state, similar to weak compatibility in [Bau+10]. Hence, incompatibilities in the sense of strong compatibility as employed in MIA may be used to discover certain inefficiencies.

Another obvious difference is that Go channels are bound to a fixed message type. That is, given a data type or a channel type T, a Go channel over T has type $\mu x.\, (!T.\, x) + (?T.\, x) + \mathsf{end}$. In practice, the strict binding to a message type may be bypassed by employing a universal Go interface as message type. However, it is well-known in the Go community that communication is then not typesafe anymore.

5.2.4. Related Work

Session types have been established since the original work of Honda [Hon93] as a typing discipline for communication sessions based on the π-calculus. A good starting point in

the large amount of session type literature is [HYC16], where the standard asynchronous multiparty session types are presented.

Bejleri and Yoshida's work on synchronous multiparty session types [BY09] is probably the closest to our work due to their support of synchronous multicast communication. The most notable differences to our theory are unsurprising for session types: they support higher order communication and delegation, which we postponed to future work, and, in particular, they employ a closed systems view with global types and pessimistic compatibility. Also, subtyping and the refinement-based specification logic that is typical for interface theories are not considered. However, there are several works that study subtyping for session types, for instance [GH05; CDY14]. Their subtyping is in the spirit of IA-refinement, but again in a pessimistic closed systems view.

Bauer et al. [BHW11] investigate a value-passing variant of MIO [Bau+10] called MIOD, where transition labels include data constraints in form of pre- and postconditions. In the semantic model, guarded input/output transition systems are employed as implementations. Simulation-based definitions of implementation and refinement are provided together with notions of composition and compatibility. These result in a compositional interface theory supporting data states in a global, possibly infinite value domain.

A similar approach has been presented by Holík et al. [HIJ15] for deterministic Interface Automata. In this model, a state consists of a location and a valuation of variables, and an action label is built out of a channel name and a value. In order to make such potentially infinite models tractable in practice, a finitely representable abstraction based on transition guards and variable assignments is defined. Refinement checking, parallel composition and quotienting may be computed on the abstract representation. While value-passing Interface Automata may represent the concrete value semantics, the abstract representation models abstract values represented by variables. Due to the usage of a single, global value domain, typed communication and type safety are not considered.

6. Conclusions and Future Research Directions

In this chapter we conclude with a summary of our findings and envision future research directions based on problems that are still open.

6.1. Conclusions

Interface theories are an established family of specification theories for concurrent, reactive systems. However, substantial practical aspects of software verification are not supported by current interface theories due to several gaps in the mathematical robustness of their semantics (Gaps 1–9 of Section 1.1). This impedes the practical utility of interface theories as foundations of software engineering methods.

The goal of this thesis was to investigate the theoretical foundations of these issues in order to make interface theories more practical by closing the above gaps. This thesis contributes the interface theory *Error-preserving Modal Interface Automata* (EMIA), and demonstrates its practical utility by providing software tools that are applied in two case studies. EMIA closes Gaps 1–7 and integrates previous approaches to software design such as MTS [LX90], interface theories [AH01a; AH05; Bau+10; Buj+15; Buj+16; BV14; Che+12; LNW07; LVF15; Rac+11] and assembly theories [HK15]. As a consequence, EMIA is a uniformly integrated specification framework that supports several common needs of software engineering, e.g., reusing components, adapting components to changed operational environments, reasoning about the compatibility of multi-component assemblies, modelling software product lines, or tracking erroneous behaviour in safety-critical systems.

EMIA extends the related interface theory *Modal Interface Automata* (MIA) [Buj+16] that was co-developed by the author. In contrast to the error-abstraction employed in established interface theories including MIA, EMIA is based on a concept of *error-preservation*, whereby its refinement preorder reflects *and* preserves fatal error states. While recent interface theories [Buj+16; Rac+11] considered the problem of how to enforce required behaviour, our finer-grained error semantics also solves the dual and previously open problem of how to forbid unwanted behaviour.

We proved that EMIA is related to IA-based interface theories via a Galois insertion of MIA into EMIA, where the Galois abstraction corresponds to error-abstraction. The Galois insertion relates error-preserving and error-abstracting interface theories at several levels. First, it allows one to translate between error-preserving and error-abstracting specifications. Second, it also relates the interface-theoretic operators of error-preserving

and error-abstracting theories, e.g., parallel composition. Third, it relates these two types of interface theories at the level of mathematical proofs, highlighting the role of error-abstraction as a monotonic operator preserving many desired properties. Although we proved this relationship explicitly for the concrete pair of theories MIA and EMIA, the approach is of more principled nature: we may equip every other theory X shown in Figure 2.4 with an error-preserving counterpart EX and a Galois insertion $(\gamma, \alpha) \colon \mathsf{X} \to \mathsf{EX}$.

EMIA's error-preservation permits one to distinguish unspecified actions that may exhibit arbitrary unknown behaviour from forbidden actions that represent unwanted behaviour. This solves several practical issues related to error-abstraction. EMIA directly supports reasoning about the compatibility of multi-component assemblies (Section 3.3.2). In contrast, previous interface theories have to resort to separate assembly theories [HK15]. The more detailed semantics provided by EMIA's error preservation is also useful when modelling software product lines (Section 3.3.3), where one is interested in finding compatible subfamilies of concrete products. Previous interface theories would consider product lines incompatible as soon as they contain at least one incompatible pair of products. The error-abstraction employed in other interface theories enables one to redefine erroneous situations to be non-erroneous and, hence, to introduce unwanted behaviour during refinement (Section 3.3.1). This is particularly undesired when specifying safety-critical systems. In contrast, EMIA's error-preservation permits one to track erroneous behaviour of safety-critical systems and to reason about possible resolutions of such errors. As a side benefit, one may employ several localised compatibility concepts simultaneously, which bridges the gap between MTS's unanimous composition and the error-aware composition of interface theories (Section 3.3.4).

Component reuse is supported by a quotient operator (Section 3.1.7 and page 60) that the author contributed to MIA and EMIA. It is the first quotient operator that permits the decomposition of *nondeterministic* interfaces. In particular, the MIA quotient takes error-abstraction in parallel composition into account (in contrast to [Rac+11]).

Alphabet extension operators (Section 3.1.9) allow one to adapt components to changed operational environments. Beyond concrete operators, we contributed a general definition of the concept of alphabet extension operators.

In Section 4.3 we demonstrated the practical utility of interface theories by employing EMIA in a case study of a safety-critical railway control system and MIA for designing a client-server application. To do so, we implemented MIA and EMIA as software tools and contrasted our implementation to existing tools implementing similar theories. The case studies also illustrate the component-based, incremental design approach supported by interface theories.

To our knowledge, EMIA is the most general interface theory to date in that it is nondeterministic rather than deterministic, supports internal behaviour, permits optimistic, pessimistic *and* unanimous compatibility with a localised compatibility concept and supports error-preservation *and* error-abstraction as well as heterogeneous specifications. The particular challenge that was solved for reaching this goal is to address all the above issues simultaneously within a single interface theory. In contrast, previous attempts to solve such gaps sacrificed other desired properties of the theory.

Further, we extended the interface theory IA to the behavioural type system ΠIA, permitting one to express the internal structure of messages. We proved standard theorems such as subject reduction and type safety. With ΠIA, we demonstrated the feasibility of developing interface theories into behavioural type theories, whereby type checking is a computationally efficient approximation of refinement checking. This is a first step towards filling the gap between the more specification-oriented interface theories and the more implementation oriented behavioural types.

6.2. Future Directions

This section summarises the author's view on open questions and describes possible future research directions. We first consider purely interface-theoretic issues and then discuss aspects related to developing interface theories into behavioural type theories. Building on more than 15 years of existing research on interface theories, we developed EMIA into an interface theory that is both mathematically robust and practical. With this result we have reached an important goal, and there are rather few purely interface-theoretic questions left. Therefore, the author believes that the next urgent research questions in this domain concern the transition from specification to implementation. As a consequence and as a result of our first step towards interface-theoretic behavioural type theories, we give more room to this second aspect.

6.2.1. Interface Theories

The quotient operator still has some restrictions with respect to internal behaviour and with respect to nondeterminism of the divisor. To our knowledge, this question is generally unsolved for interface theories. A partial solution may be found by adapting the nondeterministic quotient operator of DMTS [Ben+13]. A different but related question is how to choose the input alphabet $I_{P/\!\!/D}$ of the quotient within the interval $I_P \setminus I_D \subseteq I_{P/\!\!/D} \subseteq I_P \cup O_D$ as explained in Section 3.4.2.

As pointed out in Section 3.4.4, it is desirable to extend EMIA's support of heterogeneous specifications to include temporal logic operators as Lüttgen and Vogler do for LTS and MTS in [LV11]. Their work has already been adapted to MIA by Bujtor and Vogler in [BV16]; therefore, we consider a further adaptation to EMIA including the consideration of the Galois insertion only a minor issue.

Another question concerns the investigation of other compatibility notions necessary for modelling common communication mechanisms. When employing shared variable communication as in [Alf+05], one would consider a state $p \parallel q$ as illegal if both specify an access to a shared resource and at least one of them as an output. Concerning buffered communication, we may model buffers explicitly as EMIAs and employ EMIA's standard compatibility concept. However, when reasoning about such systems one would like to abstract from communication buffers and to model a compatibility notion that permits delaying outputs and considers components incompatible if the number of delayed messages may exceed the buffer size.

Concerning tool support, it is desirable to investigate which techniques scale well in

practice. For example, checking refinement in our Go implementation based on a transformation into quantified boolean formulas as in [Ben+15] is significantly outperformed by a transformation into boolean equation systems.

6.2.2. Behavioural Types

For simplicity, we restricted interface theories and session types to a reasonably common core. In particular, we omitted the features hiding, name passing and replication, and did not study standard interface-theoretical operations such as conjunction, disjunction and quotient. Hence, it is desirable to extend ΠIA to include the full expressiveness of EMIA. Investigating hiding should be feasible, as it is present in both session types and interface theories. However, hiding has significant differences in session types and EMIA. In session types, hiding moves channel names into a local scope but different hidden channels are still distinguishable. In contrast, EMIA's hiding operator makes all hidden actions equal such that nondeterminism may be introduced, which affects the proof of subject reduction (Proposition 5.14). Also, refinement/subtyping and the related conjunction are significantly more involved in the presence of hiding because they have to abstract from internal behaviour. To our knowledge, such an abstraction has not been considered for session types. The same is true for quotienting. Further, the strict separation of input and output alphabets common in interface theories implies that a component cannot use a channel for both input and output. When considering implementation languages such as Go, this restriction is undesired and may be removed as is done for Sociable Interfaces [Alf+05]. Further, it would be interesting to relate our behavioural type theory to value passing interface theories as presented in [BHW11] for MIO and in [HIJ15] for IA.

Concerning higher-order communication, we envision to replace $\bar{T}$ by T in Definition 5.3. This would allow one to send arbitrary objects, including processes, such that Map-Reduce-like [DG04] distributed systems could be specified. However, current interface theories do not support name passing as known from the π-calculus and session types, limiting their expressiveness to static communication networks. In order to model a train, e.g., for our case study, it is required to include the complete track model in the train model. Although possible, this approach is inconvenient because the same information has to be repeated in every train. Beyond name passing, we anticipate that the flexibility provided by the the distributed π-calculus [MPW92] or the ambient calculus [CG00] may be a good foundation for specifying such models.

The derivations (5.7) and (5.8) on page 110 define a dataflow direction that is not present in interface theories. In this regard, it would be interesting to separate the concerns of action/reaction from input/output in interface theories. Such a separation would require one to generalise interface theories beyond their traditional unique output responsibility, such that a component is permitted to use the same channel for input and output. In session types, this is mainly possible due to asynchronous communication and a strict separation of active and reactive states. With synchronous communication and mixed states, there are several possibilities of how to interpret a situation where several processes try to output on the same channel, namely synchronisation, interleaving and

incompatibility. The interleaving variant of this idea has been presented for Sociable Interfaces [Alf+05].

Several researchers have established Curry-Howard correspondences between session types and fragments of linear logic [CPT16; Car+16]. When investigating Curry-Howard correspondences between interface-based behavioural type theories and appropriate logics, these fragments will be insufficent. In particular, multicast synchronous communication is beyond a purely linear fragment of logic, and the additional operations such as conjunction and quotienting must also be considered.

Synchronous multicasting session types with unreliable communication and failure recovery are presented in [KGG14]. To our knowledge, this interesting extension has not been investigated for interface theories. We envision that failures due to unreliable communication may be modelled with EMIA's fatal error states where input transitions labelled with recovery actions represent the possibility of failure recovery.

A. Mathematical Notation

This chapter briefly summarises the mathematical concepts employed by us, in order to make this dissertation more self-contained. As these concepts can be found in many standard text books, such as [Grä79; Grä96], we mostly do not provide citations.

A.1. General Mathematical Nomenclature

Given a set S, the powerset of S is denoted as $\mathfrak{P}(S)$. The disjoint union of sets S and T is defined as $S \uplus T := (\{0\} \times S) \cup (\{1\} \times T)$. However, we often implicitly assume that S and T are disjoint and write s instead of $(0, s)$ and t instead of $(1, t)$ for elements of $S \uplus T$.

Let S be a set. A binary relation $R \subseteq S \times S$ is an *equivalence relation* if it is reflexive, symmetric and transitive. Given an element $s \in S$, the *equivalence class* $[s]_R$ of s with respect to R is defined as $[s]_R := \{t \in S \mid sRt\}$. The set of equivalence classes $S/R := \{[s]_R \mid s \in S\}$ is called the *quotient set* of S with respect to R.

A.2. Universal Algebra

A *signature* Σ is a pair $(F_\Sigma, \mathrm{ar}_\Sigma)$ consisting of a finite set $F_\Sigma := \{f_1, \ldots, f_n\}$ of *function symbols* and a function $\mathrm{ar}_\Sigma : F_\Sigma \to \mathbf{N}$ into the natural numbers that assigns each function symbol its *arity*. Given a signature Σ, a *partial Σ-algebra* $\mathcal{A}$ is a tuple $(|\mathcal{A}|, f_1^\mathcal{A}, \ldots, f_n^\mathcal{A})$, where $|\mathcal{A}|$ is a set called the *carrier set of* $\mathcal{A}$ and $f_i^\mathcal{A} : |\mathcal{A}|^{\mathrm{ar}\, f_i} \rightharpoonup |\mathcal{A}|$ for $i = 1, \ldots, n$ are partial functions. $\mathcal{A}$ is *total* if all functions $f_i^\mathcal{A}$ are total.

The set of Σ-*terms* over a set X of variables is defined as the smallest set $\mathcal{T}_\Sigma(X)$ such that $X \subseteq \mathcal{T}_\Sigma(X)$ and, if $f \in F_\Sigma$, $t_1, \ldots, t_{\mathrm{ar}\, f} \in \mathcal{T}_\Sigma(X)$, then $f(t_1, \ldots, t_{\mathrm{ar}\, f}) \in \mathcal{T}_\Sigma(X)$. A term $t \in \mathcal{T}_\Sigma(X)$ may be interpreted in a Σ-algebra $\mathcal{A}$ via a valuation $\rho\colon X \mapsto |\mathcal{A}|$ in the obvious way, written $[\![t]\!]_\rho$. If any subterm of t is undefined due to the partiality of a function, then $[\![t]\!]_\rho$ is undefined, too. Given two Σ-terms s and t, a Σ-algebra $\mathcal{A}$ *(totally) satisfies* an equation $s \equiv t$ if, for all valuations ρ, we have $[\![s]\!]_\rho = [\![t]\!]_\rho$. An equation $s \equiv t$ is *strongly satisfied* if, for all valuations ρ, if one of $[\![s]\!]_\rho$ or $[\![t]\!]_\rho$ is defined, then both are defined and equal. It is *weakly satisfied* if, for all valuations ρ, if $[\![s]\!]_\rho$ and $[\![t]\!]_\rho$ are defined, then they are equal. For example, we distinguish (total) associativity, strong associativity and weak associativity when considering the equation $(x \circ y) \circ z \equiv x \circ (y \circ z)$.

A.3. Order Theory

Let S be a set. A binary relation $R \subseteq S \times S$ is called a *preorder* if it is reflexive and transitive, i.e., if, for all $s \in S$, sRs and, for all $s, t, u \in S$, $sRt \wedge tRu \implies sRu$, respectively. If S is a set and R a preorder on S, then we call (S, R) a *preordered set*.

Any preorder R on a set S induces an equivalence relation $\equiv_R$ on S defined by $s \equiv_R s' \iff sRs' \wedge s'Rs$. In case we consider several preorders R_i, for some indices $i \in I$, we also write $\equiv_i$ instead of $\equiv_{R_i}$.

Let $(S, \sqsubseteq)$ be a preordered set. A function $f\colon S \to S$ is called *extensive* if, for all $s \in S$, $s \sqsubseteq f(s)$. Given a subset $T \subseteq S$, an element $s \in S$ is called a *lower* (*upper*) *bound* of T if, for all $t \in T$, we have $s \sqsubseteq t$ ($t \sqsubseteq s$). We say that $t \in T$ is a *least* (*greatest*) *element* of T if, for all $t' \in T$, we have $t \sqsubseteq t'$ ($t' \sqsubseteq t$). In general, the least (greatest) elements of a subset are not unique. However, they are equivalent with respect to $\equiv_\sqsubseteq$. A preordered set $(S, \sqsubseteq)$ is called a *lattice* if any finite subset of S has a least upper bound and a greatest lower bound. A lattice that is closed under least upper and greatest lower bounds of arbitrary subsets is called a *complete lattice*. A function $f\colon S \times S \to S$ that assigns any pair of elements a greatest lower (least upper) bound is called an *infimum* (*supremum*) *function*. We often write $s \sqcap s'$ ($s \sqcup s'$) instead of $f(s, s')$.

Let $(S, \sqsubseteq)$ and $(T, \preceq)$ be preordered sets. A function $f\colon S \to T$ is *monotonic* if, for all $s, s' \in S$, $s \sqsubseteq s' \implies f(s) \preceq f(s')$. A pair of monotonic functions $f\colon S \to T$ and $g\colon T \to S$ is a *Galois connection* if, for all $s \in S$ and $t \in T$, we have $s \sqsubseteq g(t) \iff f(s) \preceq t$. A Galois connection is called a *Galois insertion* if for all $t \in T$, $(f \circ g)(t) \equiv_\preceq t$.

The following paragraph on *formal concept analysis* is based on [GW99]. A *formal context* is a triple (O, A, I), where O is a set of *objects*, A is a set of *attributes* and $I \subseteq O \times A$ is a binary relation called the *incidence relation*. Given a set of objects (attributes) $S \subseteq O$ ($T \subseteq A$), we define its *derivation* by $S' := \{a \in A \mid \forall o \in S.\, oIa\}$ ($T' := \{o \in O \mid \forall a \in T.\, oIa\}$). A *formal concept* is a pair (S, T) with $S \subseteq O$ and $T \subseteq A$, such that $S' = T$ and $T' = S$. The set of all formal concepts of a formal context (O, A, I) forms a complete lattice $\mathfrak{B}(O, A, I)$, called the *concept lattice* of (O, A, I).

A.4. Mathematical Symbols

$\mathbf{N}$	set of natural numbers
Set	category of sets
$\mathfrak{P}$	powerset operator
$\mathfrak{B}$	concept lattice
$\uplus$	disjoint union
$\complement$	set-theoretic complement
ar	arity of a function
bcl	backward closure
fv	free variables
id	identity function or relation
ill	illegal states
may	set of may target states
must	set of must target states
$\lvert\cdot\rvert$	cardinality of a set, carrier set of a structure
$[\cdot]$	equivalence class, alphabet extension
$[\![\cdot]\!]$	semantics of a term

$\wedge$	logical conjunction	$\overset{a}{\rightarrowtail\!\!\!\!\twoheadrightarrow}$	weak must-transition
$\vee$	logical disjunction	$\overset{a}{\rightarrowtail\!\!\!\!\!\!\twoheadrightarrow}$	weak may-transition
$\Longrightarrow$	logical implication	$\overset{a}{\longrightarrow\!\!\!\!\twoheadrightarrow}$	trailing-weak must-transition
$\Longleftrightarrow$	logical equivalence	$\overset{a}{\dashrightarrow\!\!\!\!\twoheadrightarrow}$	trailing-weak may-transition
$\neg$	negation	$\models$	implementation relation, satisfaction relation
$\sqcap$	specification-theoretic conjunction		
$\sqcup$	specification-theoretic disjunction	$\equiv$	equivalence, mutual refinement
$\rightarrow$	specification-theoretic implication	$\#$	complexity
$/$	hiding	$\sqsubseteq$	refinement
$\backslash$	restriction	$\sqsubseteq_{\mathrm{e}}$	error-preserving refinement
$\otimes$	parallel product	$\sqsubseteq_{\mathrm{m}}$	error-abstracting refinement
$\parallel$	parallel composition	$\sqsubseteq_{\mathsf{mts}}$	modal refinement
$\mid$	parallel composition with immediate hiding	$\preceq$	subtype relation, assembly refinement
$\&$	conjunctive product	$\top$	universal state
$/\!/$	quotient operator	$\vdash$	entailment
$\oslash$	pre-quotient operator	$:$	has-type relation
$\overset{a}{\longrightarrow}$	must-transition	φ	assembly encapsulation
$\overset{a}{\dashrightarrow}$	may-transition	$[x/y]$	substitution of x for y

B. Source Code of Gemia

This Chapter lists the source code of Gemia, which consists of three Haskell modules. Module Gemia implements the core data structure for MIAs and the operations on MIAs. Module Aux implements some general purpose data structures and auxiliary functions. Module Graphviz provides an API for drawing transition systems with Graphviz.

B.1. Module Gemia

```
module Gemia where

import Data.List
import Data.Ord
import Data.Function
import Graphviz
import Aux

-- Status of a node (errors, inconsistencies)
data Status = Status { err::Bool, incons::Bool } deriving Show

normalStatus = Status False False

-- Tail of a transition (action and targets)
data Tail n a = Tail { action::a, targets::[n] } deriving Show

-- A node with its status all its tails
data TransitionSet n a =
  TransitionSet { source::n, status::Status, actions::[Tail n a] } deriving Show

data TransitionSystem n a =
  TransitionSystem { start::n, trans::[TransitionSet n a]} deriving Show

-- All nodes of a transition system
nodes :: Eq n => TransitionSystem n a -> [n]
nodes tsys = nub (start tsys : concatMap f (trans tsys))
  where f tset = (source tset : concatMap targets (actions tset))

-- Remove nodes from a tail
rmNodesTail :: Eq n => [n] -> Tail n a -> Tail n a
rmNodesTail ns t = Tail (action t) ((targets t)\\ns)

-- Remove nodes from a transition set
rmNodesTransitionSet :: Eq n => [n] -> TransitionSet n a -> TransitionSet n a
rmNodesTransitionSet ns t = TransitionSet (source t) (status t) actions'
  where actions' = map (rmNodesTail ns) (actions t)
```

```
-- Remove nodes from a transition system
rmNodesTransitionSystem :: Eq n =>
  [n] -> TransitionSystem n a -> TransitionSystem n a
rmNodesTransitionSystem ns t = TransitionSystem (start t) trans'
  where trans' = map (rmNodesTransitionSet ns) trans''
        trans'' = filter (flip notElem ns . source) (trans t)

-- Remove empty tails
rmEmptyTails :: TransitionSet n a -> TransitionSet n a
rmEmptyTails ts = TransitionSet (source ts) (status ts) actions'
  where actions' = filter (not . null . targets) (actions ts)

-- An action has a modality, an IO-type and a label
data Action m k l = Action { modality::m, ioType::k, label::l } deriving (Show)

-- Concrete instance for modalities
data Modality = Required | Optional deriving (Eq, Ord, Show)

-- Concrete instance for IO-types
data IOType = Input | Output deriving (Eq, Ord, Show)

-- Concrete instance for nodes
type Node n = BTree n

-- Product of nodes
nprod :: Node n -> Node n -> Node n
nprod = Branch

-- All pairs of nodes
npairs = makePairsBy nprod

setErrorStatus :: TransitionSet n a -> TransitionSet n a
setErrorStatus t = TransitionSet (source t) newStatus (actions t)
  where newStatus = Status True (incons $ status t)

setErrorStates :: (Eq n) => TransitionSystem n a -> [n] -> TransitionSystem n a
setErrorStates t ns = TransitionSystem (start t) newTransitions
  where
    newTransitions  = map changeStatus (trans t)
    changeStatus tr = if (source tr) `elem` ns then setErrorStatus tr else tr

-- General type for modal input-output transition systems
type GemiaLabel    = String
type GemiaAction   = Action Modality IOType GemiaLabel
type GemiaTail n   = Tail (Node n) GemiaAction
type GemiaTrans n  = TransitionSet (Node n) GemiaAction
type Gemia n       = TransitionSystem (Node n) GemiaAction

-- insert b at position a in [c] which is given by cmp with insert-function ins
insertByWith :: (a -> a -> Bool) -> (a -> a -> a) -> [a] -> a -> [a]
insertByWith _ ins [] a = [a]
insertByWith cmp ins (b:bs) a = if cmp a b then b':bs else b:bs'
```

```
  where b'  = ins a b
        bs' = insertByWith cmp ins bs a

-- Add a transition
addTrans :: Eq n => Gemia n -> GemiaTrans n -> Gemia n
addTrans g t = g { trans = trans' }
  where
    trans' = foldl addNode (insertByWith cmp ins (trans g) t) newnodes
    newnodes = concatMap targets $ actions t
    addNode ts n = insertByWith cmp ins ts (TransitionSet n normalStatus [])
    cmp      = (==) `on` source
    ins x y  = TransitionSet (source x) (status x) (((++) `on` actions) x y)

-- Make a gemia from initial state and list of transitions
makeGemia :: Eq n => Node n -> [GemiaTrans n] -> Gemia n
makeGemia i = foldl addTrans (TransitionSystem i [])

-- Make a gemia from initial state, list of transitions and list of error states
makeGemiaErr :: Eq n => Node n -> [GemiaTrans n] -> [Node n] -> Gemia n
makeGemiaErr i ts = setErrorStates (foldl addTrans (TransitionSystem i []) ts)

-- Make gemia tail
makeTail :: Modality -> IOType -> GemiaLabel -> [Node n] -> GemiaTail n
makeTail m k l ns = Tail (Action m k l) ns

-- Make gemia transition
makeTrans :: Modality -> Node n -> IOType -> GemiaLabel -> [Node n] ->
  GemiaTrans n
makeTrans m n k l ns = TransitionSet n normalStatus [makeTail m k l ns]

-- Make gemia required transition
req :: n -> IOType -> GemiaLabel -> [n] -> GemiaTrans n
req n k l ns = makeTrans Required (Leaf n) k l (map Leaf ns)

-- Make gemia optional transition
opt :: n -> IOType -> GemiaLabel -> n -> GemiaTrans n
opt  n k l t = makeTrans Optional (Leaf n) k l [Leaf t]

-- Transform Gemias to Dot-language
instance Dotable GemiaLabel where
  toDot = id

instance Dotable IOType where
  toDot Input  = "?"
  toDot Output = "!"

instance Dotable n => Dotable (GemiaTrans n) where
  toDot ts = src ++ tls
    where s    = toDot (source ts)
          src  = if null srcOptions then "" else makeNode s srcOptions
          srcOptions = incOpt ++ errOpt
          incOpt = if (incons $ status ts)
                 then ["color=\"#888888\"","fontcolor=\"#888888\""]
```

```
               else []
          errOpt = if (err $ status ts)
                   then ["shape=hexagon"]
                   else []
          as   = actions ts
          tls  = concatMap (arrow s) as
          arrow s a = if length trgs == 1
            then simpleArrow s l (head trgs) tlOptions
            else multiArrow s l trgs tlOptions
              where
                trgs = map toDot (targets a)
                l    = (toDot $ label $ action a) ++ (toDot $ ioType $ action a)
                modOpt = if (modality $ action a) == Optional
                         then ["style=dashed"]
                         else []
                tlOptions = modOpt ++ incOpt

instance Dotable n => Dotable (Gemia n) where
  toDot g = "digraph {\n" ++ startNode ++ transitions ++ "}\n"
    where startNode = "\"" ++ toDot (start g) ++ "\"[shape=box,style=rounded]\n"
          transitions = concatMap toDot (trans g)

-- All action labels of a Gemia
labels :: Gemia n -> [GemiaLabel]
labels = nub . (map (label . action)) . (concatMap actions) . trans

-- All actions of a Gemia by label
actionsByLabel :: Gemia n -> [GemiaAction]
actionsByLabel =
  nubBy ((==) `on` label) . (map action) . (concatMap actions) . trans

-- All targets of a node reachable
reachableLocally :: Eq n => [GemiaTrans n] -> Node n -> [Node n]
reachableLocally ts n = nub ns
  where as = concatMap actions [t|t<-ts, source t == n]
        ns = concatMap targets as

-- All targets of a node reachable by actions with label l
reachableLocallyLabel :: Eq n => [GemiaTrans n] -> Node n -> GemiaLabel ->
  [Node n]
reachableLocallyLabel ts n l = nub ns
  where as = concatMap actions [t|t<-ts, source t == n]
        ns = concatMap targets (filter ((==l) . label . action) as)

reachableAux :: Eq n => [GemiaTrans n] -> ([Node n],[Node n]) ->
  ([Node n],[Node n])
reachableAux _ (done, []) = (done, [])
reachableAux ts (done, t:todo) = reachableAux ts (done', todo')
  where done' = t:done
        todo' = todo ++ (((reachableLocally ts t)\\done')\\todo)

reachable :: Eq n => [GemiaTrans n] -> Node n -> [Node n]
reachable ts n = fst (reachableAux ts ([],[n]))
```

```
-- prune empty tails
pruneEmptyTails :: Gemia n -> Gemia n
pruneEmptyTails g = TransitionSystem (start g) trans'
  where trans' = map rmEmptyTails (trans g)

-- prune unreachable states
pruneUnreachable :: Eq n => Gemia n -> Gemia n
pruneUnreachable g = rmNodesTransitionSystem unreachable g
  where unreachable = (nodes g)\\(reachable (trans g) (start g))

-- prune inconsistent states (initial state is not pruned!)
pruneIncons :: Eq n => Gemia n -> Gemia n
pruneIncons g = if null currentIncons then pruneEmptyTails g else pruneIncons g'
  where currentIncons = map source $ filter f (trans g)
        f = not . null . (filter incons) . actions
        incons a = ((==Required) . modality $ action a) && (null $ targets a)
        g' = rmNodesTransitionSystem currentIncons g


-- Alphabet extension
addLoops :: Eq n => Gemia n -> [GemiaAction] -> Gemia n
addLoops g as = foldl addTrans g (concatMap makeloops (nodes g))
  where makeloops n = map (\a->TransitionSet n normalStatus [Tail a [n]]) as

-- Extend gemias by their mutually foreign actions
extendMutually :: Eq n =>
  (Modality -> Modality) -> (Modality -> Modality) ->
  (IOType -> IOType) -> (IOType -> IOType) ->
  Gemia n -> Gemia n -> (Gemia n, Gemia n)
extendMutually m1 m2 iot1 iot2 g1 g2 =
  (addLoops g1 actions1, addLoops g2 actions2)
  where native1  = actionsByLabel g1
        native2  = actionsByLabel g2
        foreign1 = deleteFirstsBy ((==) `on` label) native2 native1
        foreign2 = deleteFirstsBy ((==) `on` label) native1 native2
        actions1 = map (updateAction m1 iot1) foreign1
        actions2 = map (updateAction m2 iot2) foreign2
        updateAction m iot a = Action (m $ modality a) (iot $ ioType a) (label a)

-- Auxiliary function for operations where systems should not be extended
noExtend x y = (x,y)

-- Meta product
metaprod :: Eq n => (GemiaTrans n -> GemiaTrans n -> GemiaTrans n) ->
  (Gemia n -> Gemia n -> (Gemia n, Gemia n)) -> Gemia n -> Gemia n -> Gemia n
metaprod prodTS extend g1 g2 = TransitionSystem start12 trans12
  where
    start12   = nprod (start g1') (start g2')
    trans12   = [prodTS n1 n2|n1<-trans g1', n2<-trans g2']
    (g1',g2') = extend g1 g2

-- parallel product with alphabet extension
```

```
pprod :: Eq n => Gemia n -> Gemia n -> Gemia n
pprod = metaprod pprodTS extend
  where
    extend = extendMutually cr cr id id
    cr = const Required

-- local parallel product
pprodTS :: Eq n => GemiaTrans n -> GemiaTrans n -> GemiaTrans n
pprodTS ts1 ts2 = TransitionSet source12 status12 acts12
  where
    src1     = source ts1
    src2     = source ts2
    source12 = nprod src1 src2
    status12 = Status error12 incons12
    error12  = (err $ status ts1) || (err $ status ts2)
    incons12 = (incons $ status ts1) || (incons $ status ts2)
    acts12   = [comb tl1 tl2|tl1<-actions ts1, tl2<-actions ts2,
      (label $ action tl1)==(label $ action tl2)]
    comb tl1 tl2 = Tail (Action m k l) trgs
      where m = max (modality $ action tl1) (modality $ action tl2)
            k = max (ioType $ action tl1) (ioType $ action tl2)
            l = label $ action tl1
            trgs = makePairsBy nprod (targets tl1) (targets tl2)

-- pseudo quotient without compatibility
iquot :: Eq n => Gemia n -> Gemia n -> Gemia n
iquot = metaprod iquotTS extend
  where
    extend = extendMutually cr cr id (const Input)
    cr     = const Required

-- local pseudo quotient
iquotTS :: Eq n => GemiaTrans n -> GemiaTrans n -> GemiaTrans n
iquotTS ts1 ts2 = TransitionSet source12 status12 acts12
  where
    source12 = nprod (source ts1) (source ts2)
    status12 = Status False False
    acts12   = [comb tl1 tl2 | tl1<-actions ts1, tl2<-actions ts2,
      (label $ action tl1) == (label $ action tl2)]
    comb tl1 tl2 = Tail act trgs
      where
        act = Action (m m1 m2) (k k1 k2) (label $ action tl1)
        m1  = modality $ action tl1
        m2  = modality $ action tl2
        m mA mB = if k2 == Output then Required else max mA mB
        k1  = ioType $ action tl1
        k2  = ioType $ action tl2
        k kA kB = if kA == kB then Input else Output
        trgs = makePairsBy nprod (targets tl1) (targets tl2)

-- Required and optional actions
reqOnly, optOnly :: [GemiaTail n] -> [GemiaTail n]
reqOnly = filter ((==Required) . modality . action)
```

```
optOnly = filter ((==Optional) . modality . action)

-- conjunctive product with alphabet extension
cprod :: Eq n => Gemia n -> Gemia n -> Gemia n
cprod = metaprod cprodTS extend
  where
    extend = extendMutually co co id id
    co     = const Optional

-- local conjunctive product
cprodTS :: Eq n => GemiaTrans n -> GemiaTrans n -> GemiaTrans n
cprodTS tsA tsB = TransitionSet sourceAB statusAB actsAB
  where
    sourceAB = nprod (source tsA) (source tsB)
    statusAB = Status errorAB inconsAB
    errorAB  = (err $ status tsA) && (err $ status tsB)
    inconsAB = (incons $ status tsA) || (incons $ status tsB) || localIncons
    localIncons = not $ null (filter (null . targets) reqs)
    actsAB   = reqs ++ opts
    reqs     = (map (f nprod tsB) reqsA) ++ (map (f (flip nprod) tsA) reqsB)
    reqsA    = reqOnly (actions tsA)
    reqsB    = reqOnly (actions tsB)
    f comb ts2 tail1 = Tail (action tail1)
      [comb trg1 trg2 | trg1<-targets tail1,
        trg2<-reachableLocallyLabel [ts2] (source ts2) (label $ action tail1)]
    optsA    = optOnly (actions tsA)
    optsB    = optOnly (actions tsB)
    opts     = [Tail (action tailA)
                 (makePairsBy nprod (targets tailA) (targets tailB)) |
                 tailA<-optsA, tailB<-optsB,
                 (label $ action tailA) == (label $ action tailB)]
```

B.2. Module Aux

```
module Aux where

import Graphviz

-- Binary trees
data BTree a = Leaf a | Branch (BTree a) (BTree a) deriving (Eq,Show)

instance (Ord a) => Ord (BTree a) where
  compare (Leaf a) (Leaf b) = compare a b
  compare (Branch a1 a2) (Branch b1 b2)
    | compare a1 b1 == LT = LT
    | compare a1 b1 == EQ = compare a2 b2
    | otherwise           = GT

instance (Dotable n) => Dotable (BTree n) where
  toDot (Leaf a) = toDot a
  toDot (Branch a b) = "(" ++ (toDot a) ++ "," ++ (toDot b) ++ ")"
```

```
-- List of pairs
makePairsBy :: (a -> b -> c) -> [a] -> [b] ->[c]
makePairsBy f as bs = [f a b|a<-as,b<-bs]

-- List of all possible choices of elements from a list of lists
choices :: [[a]] -> [[a]]
choices [] = [[]]
choices (x:xs) = concatMap (\y -> prependAll y (choices xs)) x

-- Prepend an element to each list in a list of list
prependAll :: a -> [[a]] -> [[a]]
prependAll x ys = map (x:) ys
```

B.3. Module Graphiz

```
module Graphviz where

import Data.List

class Dotable a where
  toDot :: a -> String

instance Dotable Int where
  toDot i = show i

-- make a node from name and options
makeNode :: String -> [String] -> String
makeNode name options =
  "\"" ++ name ++ "\" [" ++ intercalate "," options ++ "]\n"

-- make an arrow from source, target and options
makeArrow :: String -> String -> [String] -> String
makeArrow source target options =
  "\"" ++ source ++ "\"->\"" ++ target ++ "\" [" ++ intercalate "," options ++
  "]\n"

-- make a labelled arrow
simpleArrow :: String -> String -> String -> [String] -> String
simpleArrow s l t opts =
  makeArrow s t (("label=\""++ l ++ "\""):opts)

-- make a labelled multiarrow
multiArrow :: String -> String -> [String] -> [String] -> String
multiArrow s l ts opts = (makeNode dummy ["shape=point"])
  ++ (makeArrow s dummy ("arrowhead=none":("label=\"" ++ l ++ "\""):opts))
  ++ (concatMap (\t->makeArrow dummy t opts) ts)
  where dummy = intercalate "-" (s:l:ts)
```

C. Source Code of the Case Studies

This chapter lists the source code of the two case studies presented in Chapter 4.

C.1. Client-Server Case Study

We implemented the client-server case study in Gemia's embedded DSL, i.e., the case-study itself is a valid Haskell program.

```
import Gemia
import Data.List
import System.Environment
import System.IO

import Aux
import Graphviz

-- Global server specification.
global :: Gemia Int
global = makeGemia (Leaf 0)
  [req 0 Input  "rqst" [1],
   opt 1 Output "fail" 0,
   opt 1 Output "resp" 0]

-- back-end 1: local cache
cache :: Gemia Int
cache = makeGemia (Leaf 0)
  [req 0 Input  "rqst1" [1],
   req 1 Output "resp1" [0],
   opt 1 Output "miss" 0]

-- back-end 2: remote database
database :: Gemia Int
database = makeGemia (Leaf 0)
  [req 0 Input "rqst2" [1],
   req 1 Output "resp2" [0]]

-- R1: forward to one of the back-ends
r1 :: Gemia Int
r1 = makeGemia (Leaf 0)
  [opt 0 Input  "rqst" 1,
   opt 2 Output "rqst1" 0,
   opt 2 Output "rqst2" 0,
   req 1 Output "sel" [2],
   opt 0 Output "resp" 0,
   opt 0 Output "rqst1" 0,
   opt 0 Output "rqst2" 0]

-- R2: one back-end or both
r2 :: Gemia Int
r2 = makeGemia (Leaf 0)
  [opt 0 Output "sel" 1,
   opt 0 Output "sel" 2,
   req 1 Output "rqst1" [0],
   req 2 Output "rqst2" [0],
   opt 0 Output "resp" 0,
   opt 0 Output "rqst1" 0,
   opt 0 Output "rqst2" 0]

-- R3: after rqst1, first wait for resp1,
--     then respond to client
r3 :: Gemia Int
r3 = makeGemia (Leaf 0)
  [opt 0 Output "rqst1" 1,
   req 1 Input  "resp1" [2],
   req 1 Input  "miss" [0],
   req 2 Output "resp" [0],
   opt 0 Output "resp" 0]

-- R4: after rqst2, first wait for resp2,
--     then respond to client
r4 :: Gemia Int
r4 = makeGemia (Leaf 0)
  [opt 0 Output "rqst2" 1,
   req 1 Input  "resp2" [2],
   req 2 Output "resp" [0],
   opt 0 Output "resp" 0]

-- R5: after a miss redirect to database
r5 :: Gemia Int
r5 = makeGemia (Leaf 0)
```

```
  [opt 0 Input "miss" 1,
   opt 1 Output "fbck" 2,
   opt 1 Output "fail" 0,
   req 2 Output "rqst2" [0],
   opt 0 Output "rqst2" 0,
   opt 0 Output "resp" 0]

-- Closed system
one :: Gemia Int
one = makeGemia (Leaf 0)
  (map (\x->opt 0 Output x 0)
       ["rqst","resp","fail"])

-- Conjunction of requirements,
-- unpruned and pruned.
ru, r :: Gemia Int
ru = foldl1 cprod [r1,r2,r3,r4,r5]
r  = pruneUnreachable (pruneIncons ru)

-- Conjunction of R1 and R2.
r12 :: Gemia Int
r12 = cprod r1 r2

-- foreign actions
fa = map (Action Optional Output)
  ["rqst1", "rqst2", "resp1", "resp2",
   "miss"]

-- extended global specification
global' = addLoops global fa
```

```
-- upper bound for frontend
uf :: Gemia Int
uf = global' `iquot`
  (cache `pprod` database)

-- final front-end specification
f :: Gemia Int
f = pruneUnreachable $ pruneIncons $
  uf `cprod` r

-- server
s :: Gemia Int
s = pruneUnreachable $ f `pprod` cache
  `pprod` database

-- F' hides back-end communication in F
f' = makeGemia (Leaf 0)
  [req 0 Input  "rqst" [1],
   req 1 Output "resp" [0],
   opt  1 Output "fail" 0
  ]

-- Client
client :: Gemia Int
client =  one `iquot` f'

-- Full system
sys :: Gemia Int
sys = pruneUnreachable $ client `pprod` s
```

C.2. Railway Case Study

We conducted the railway case study within our tool MiaGo. Due to the usage of an external DSL we split up the presentation into the component specifications and the system construction.

C.2.1. Component Specifications

The components of the railway case study are specified in an external DSL.

```
specification SignalUni {
  states { u }
  inis   { u }
  unis   { u }
  channels { }
  u { }
};

specification SignalUniExt {
  states { u }
  inis   { u }
  unis   { u }
  channels { go, stop }
  u { }
};

specification SignalRef1 {
  states { s, g, u }
  inis   { s }
  unis   { u }
```

```
  channels { go, stop }
  s {
    go?? -> g
    stop? -> u
  }
  g {
    stop?? -> s
    go? -> u
  }
};

specification SignalRef2 {
  states { s, g, e }
  inis   { s }
  errs   { e }
  channels { go, stop }
  s {
    go?? -> g
    stop?? -> { s, e }
  }
  g {
    stop?? -> s
    go?? -> { g, e }
  }
};

specification SignalRef3A {
  states { s, g }
  inis   { s }
  channels { go, stop }
  s {
    go?? -> g
    stop?? -> { s }
  }
  g {
    stop?? -> s
    go?? -> { g }
  }
};

specification SignalRef3B {
  states { s, g, e }
  inis   { s }
  errs   { e }
  channels { go, stop }
  s {
    go?? -> g
    stop?? -> { e }
  }
  g {
    stop?? -> s
    go?? -> { e }
  }
};

specification SignalRef3A0 {
  states { s0, g0 }
  inis   { s0 }
  channels { go0, stop0 }
  s0 {
    go0?? -> g0
    stop0?? -> { s0 }
  }
  g0 {
    stop0?? -> s0
    go0?? -> { g0 }
  }
};

specification SignalRef3A1 {
  states { s1, g1 }
  inis   { s1 }
  channels { go1, stop1 }
  s1 {
    go1?? -> g1
    stop1?? -> { s1 }
  }
  g1 {
    stop1?? -> s1
    go1?? -> { g1 }
  }
};

specification SignalRef3B0 {
  states { s0, g0, e0 }
  inis   { s0 }
  errs   { e0 }
  channels { go0, stop0 }
  s0 {
    go0?? -> g0
    stop0?? -> { e0 }
  }
  g0 {
    stop0?? -> s0
    go0?? -> { e0 }
  }
};

specification SignalRef3B1 {
  states { s1, g1, e1 }
  inis   { s1 }
  errs   { e1 }
  channels { go1, stop1 }
  s1 {
    go1?? -> g1
    stop1?? -> { e1 }
```

```
  }
  g1 {
    stop1?? -> s1
    go1?? -> { e1 }
  }
};

specification Signal3 {
  states { s3, g3, sl3 }
  inis   { s3 }
  channels { go3, stop3, slow3 }
  s3 {
    go3?? -> g3
    stop3?? -> s3
    slow3?? -> sl3
  }
  g3 {
    stop3?? -> s3
    go3?? -> g3
    slow3?? -> sl3
  }
  sl3 {
    stop3?? -> s3
    go3??   -> g3
    slow3?? -> sl3
  }
};
specification Global1 {
  states { u }
  inis   { u }
  channels { go0, go1, stop0, stop1 }
  u {
    go0! -> u
    go1! -> u
    stop0! -> u
    stop1! -> u
  }
};

specification Global2 {
  states { u }
  inis   { u }
  channels { go0, go1, stop0, stop1,
    norm, alt }
  u {
    go0! -> u
    go1! -> u
    stop0! -> u
    stop1! -> u
    norm! -> u
    alt! -> u
  }
};
```

```
specification GlobalL {
  states { u }
  inis   { u }
  channels { go0, stop0, go1, stop1, go3,
    stop3, slow3, block0, free0, block1,
    free1, block2, free2, block3, free3,
    norm2, alt2 }
  u {
    go0! -> u
    stop0! -> u
    go1! -> u
    stop1! -> u
    go3! -> u
    stop3! -> u
    slow3! -> u
    block0! -> u
    free0! -> u
    block1! -> u
    free1! -> u
    block2! -> u
    free2! -> u
    block3! -> u
    free3! -> u
    norm2! -> u
    alt2! -> u
  }
};
specification RqContr {
  states { s, g0, g1, e }
  inis   { s }
  errs   { e }
  channels { go0, go1, stop0, stop1 }
  s {
    go0! -> g0
    go1! -> g1
    stop0! -> s
    stop1! -> s
  }
  g0 {
    go0! -> g0
    go1! -> e
    stop0! -> s
    stop1! -> g0
  }
  g1 {
    go0! -> e
    go1! -> g1
    stop0! -> g1
    stop1! -> s
  }
};
```

```
specification RqS0S1 {
  states { s0s1, s0g1, g0s1, g0g1 }
  inis { s0s1 }
  errs { g0g1 }
  channels { go0, go1, stop0, stop1 }
  s0s1 {
    go0?? -> { g0s1 }
    go1?? -> { s0g1 }
    stop0?? -> { s0s1 }
    stop1?? -> { s0s1 }
  }
  g0s1 {
    go0?? -> { g0s1 }
    go1?? -> { g0g1 }
    stop0?? -> { s0s1 }
    stop1?? -> { g0s1 }
  }
  s0g1 {
    go0?? -> { g0g1 }
    go1?? -> { s0g1 }
    stop0?? -> { s0g1 }
    stop1?? -> { s0s1 }
  }
  g0g1 {
  }
};

specification RqS0YExt {
  states { s0n, g0n, s0a, g0a }
  inis   { s0n }
  errs   { g0n }
  channels { go0, stop0, norm, alt,
    go1, stop1 }
  s0n {
    go0?? -> g0n
    stop0?? -> { s0n, g0n }
    norm?? -> { s0n, g0n }
    alt?? -> { s0a, g0n }
    go1?? -> { s0n, g0n }
    stop1?? -> { s0n, g0n }
  }
  g0n { }
  s0a {
    go0?? -> g0a
    stop0?? -> { s0a, g0n }
    norm?? -> { s0n, g0n }
    alt?? -> { s0a, g0n }
    go1?? -> { s0a, g0n }
    stop1?? -> { s0a, g0n }
  }
  g0a {
    go0?? -> { g0a, g0n }
    stop0?? -> { s0a, g0n }
    norm?? -> g0n
    alt?? -> { g0a, g0n }
    go1?? -> { g0a, g0n }
    stop1?? -> { g0a, g0n }
  }
};

specification RqS1YExt {
  states { s1n, g1n, s1a, g1a }
  inis   { s1n }
  errs   { g1a }
  channels { go1, stop1, norm, alt,
    go0, stop0 }
  s1n {
    go1?? -> { g1n, g1a }
    stop1?? -> { s1n, g1a }
    norm?? -> { s1n, g1a }
    alt?? ->  { s1a, g1a }
    go0?? -> { s1n, g1a }
    stop0?? -> { s1n, g1a }
  }
  g1n {
    go1?? -> { g1n, g1a }
    stop1?? -> { s1n, g1a }
    norm?? -> { g1n, g1a }
    alt?? -> g1a
    go0?? -> { g1n, g1a }
    stop0?? -> { g1n, g1a }
  }
  s1a {
    go1?? -> g1a
    stop1?? -> { s1a, g1a }
    norm?? -> { s1n, g1a }
    alt?? -> { s1a, g1a }
    go0?? -> { s1a, g1a }
    stop0?? -> { s1a, g1a }
  }
  g1a { }
};

specification RqSignalsL {
  states { s, g0, g1, g3, sl3, e }
  inis   { s }
  errs   { e }
  channels { go0, go1, go3, stop0,
    stop1, stop3, slow3 }
  s {
    go0! -> g0
    go1! -> g1
    go3! -> g3
    stop0! -> s
    stop1! -> s
    stop3! -> s
```

```
    slow3! -> sl3
  }
  g0 {
    go0! -> g0
    go1! -> e
    go3! -> e
    slow3! -> e
    stop0! -> s
    stop1! -> g0
    stop3! -> g0
  }
  g1 {
    go0! -> e
    go1! -> g1
    go3! -> e
    slow3! -> e
    stop0! -> g1
    stop1! -> s
    stop3! -> g1
  }
  g3 {
    go0! -> e
    go1! -> e
    go3! -> g3
    slow3! -> sl3
    stop0! -> g3
    stop1! -> g3
    stop3! -> s
  }
  sl3 {
    go0! -> e
    go1! -> e
    go3! -> g3
    slow3! -> sl3
    stop0! -> sl3
    stop1! -> sl3
    stop3! -> s
  }
};

specification RqS3B0Y2L {
  states { s3f0n2, s3b0n2, s3f0a2, s3b0a2,
    g3f0n2, g3b0n2, g3f0a2, g3b0a2 }
  inis   { s3f0n2 }
  errs   { g3b0a2 }
  channels { free0, block0, norm2, alt2,
    go3, stop3, slow3 }
  s3f0n2 {
    free0!  -> s3f0n2
    block0! -> s3b0n2
    norm2!  -> s3f0n2
    alt2!   -> s3f0a2
    go3!    -> g3f0n2
    slow3!  -> g3f0n2
    stop3!  -> s3f0n2
  }
  s3b0n2 {
    free0!  -> s3f0n2
    block0! -> s3b0n2
    norm2!   -> s3b0n2
    alt2!    -> s3b0a2
    go3!    -> g3b0n2
    slow3!  -> g3b0n2
    stop3!  -> s3b0n2
  }
  s3f0a2 {
    free0!  -> s3f0a2
    block0! -> s3b0a2
    norm2!   -> s3f0n2
    alt2!    -> s3f0a2
    go3!    -> g3f0a2
    slow3!  -> g3f0a2
    stop3!  -> s3f0a2
  }
  s3b0a2 {
    free0!  -> s3f0a2
    block0! -> s3b0a2
    norm2!   -> s3b0n2
    alt2!    -> s3b0a2
    go3!    -> g3b0a2
    slow3!  -> g3b0a2
    stop3!  -> s3b0a2
  }
  g3f0n2 {
    free0!  -> g3f0n2
    block0! -> g3b0n2
    norm2!   -> g3f0n2
    alt2!    -> g3f0a2
    go3!    -> g3f0n2
    slow3!  -> g3f0n2
    stop3!  -> s3f0n2
  }
  g3b0n2 {
    free0!  -> g3f0n2
    block0! -> g3b0n2
    norm2!   -> g3b0n2
    alt2!    -> g3b0a2
    go3!    -> g3b0n2
    slow3!  -> g3b0n2
    stop3!  -> s3b0n2
  }
  g3f0a2 {
    free0!  -> g3f0a2
    block0! -> g3b0a2
    norm2!   -> g3f0n2
    alt2!    -> g3f0a2
```

```
    go3!    -> g3f0a2
    slow3!  -> g3f0a2
    stop3!  -> s3f0a2
  }
  g3b0a2 {
  }
};

specification RqS3B1Y2L {
  states { s3f1n2, s3b1n2, s3f1a2, s3b1a2,
    g3f1n2, g3b1n2, g3f1a2, g3b1a2 }
  inis   { s3f1n2 }
  errs   { g3b1n2 }
  channels { free1, block1, norm2, alt2,
    go3, stop3, slow3 }
  s3f1n2 {
    free1!  -> s3f1n2
    block1! -> s3b1n2
    norm2!  -> s3f1n2
    alt2!   -> s3f1a2
    go3!    -> g3f1n2
    slow3!  -> g3f1n2
    stop3!  -> s3f1n2
  }
  s3b1n2 {
    free1!  -> s3f1n2
    block1! -> s3b1n2
    norm2!   -> s3b1n2
    alt2!    -> s3b1a2
    go3!    -> g3b1n2
    slow3!  -> g3b1n2
    stop3!  -> s3b1n2
  }
  s3f1a2 {
    free1!  -> s3f1a2
    block1! -> s3b1a2
    norm2!   -> s3f1n2
    alt2!    -> s3f1a2
    go3!    -> g3f1a2
    slow3!  -> g3f1a2
    stop3!  -> s3f1a2
  }
  s3b1a2 {
    free1!  -> s3f1a2
    block1! -> s3b1a2
    norm2!   -> s3b1n2
    alt2!    -> s3b1a2
    go3!    -> g3b1a2
    slow3!  -> g3b1a2
    stop3!  -> s3b1a2
  }
  g3f1n2 {
    free1!  -> g3f1n2
    block1! -> g3b1n2
    norm2!   -> g3f1n2
    alt2!    -> g3f1a2
    go3!    -> g3f1n2
    slow3!  -> g3f1n2
    stop3!  -> s3f1n2
  }
  g3b1n2 { }
  g3f1a2 {
    free1!  -> g3f1a2
    block1! -> g3b1a2
    norm2!   -> g3f1n2
    alt2!    -> g3f1a2
    go3!    -> g3f1a2
    slow3!  -> g3f1a2
    stop3!  -> s3f1a2
  }
  g3b1a2 {
    free1!  -> g3f1a2
    block1! -> g3b1a2
    norm2!   -> g3b1n2
    alt2!    -> g3b1a2
    go3!    -> g3b1a2
    slow3!  -> g3b1a2
    stop3!  -> s3b1a2
  }
};

specification RqB2S0L {
  states { f2s0, f2g0, b2s0, b2g0 }
  inis   { f2s0 }
  errs   { b2g0 }
  channels { block2, free2, stop0, go0 }
  f2s0 {
    block2! -> b2s0
    free2!  -> f2s0
    stop0!  -> f2s0
    go0!    -> f2g0
  }
  f2g0 {
    block2! -> b2g0
    free2!  -> f2g0
    stop0!  -> f2s0
    go0!    -> f2g0
  }
  b2s0 {
    block2! -> b2s0
    free2!  -> f2s0
    stop0!  -> b2s0
    go0!    -> b2g0
  }
  b2g0 { }
};
```

```
specification RqB2S1L {
  states { f2s1, f2g1, b2s1, b2g1 }
  inis   { f2s1 }
  errs   { b2g1 }
  channels { block2, free2, stop1, go1 }
  f2s1 {
    block2! -> b2s1
    free2!  -> f2s1
    stop1!  -> f2s1
    go1!    -> f2g1
  }
  f2g1 {
    block2! -> b2g1
    free2!  -> f2g1
    stop1!  -> f2s1
    go1!    -> f2g1
  }
  b2s1 {
    block2! -> b2s1
    free2!  -> f2s1
    stop1!  -> b2s1
    go1!    -> b2g1
  }
  b2g1 { }
};

specification RqB2S3L {
  states { f2s3, f2g3, b2s3, b2g3 }
  inis   { f2s3 }
  errs   { b2g3 }
  channels { block2, free2, stop3,
    go3, slow3 }
  f2s3 {
    block2! -> b2s3
    free2!  -> f2s3
    stop3!  -> f2s3
    go3!    -> f2g3
    slow3!  -> f2g3
  }
  f2g3 {
    block2! -> b2g3
    free2!  -> f2g3
    stop3!  -> f2s3
    go3!    -> f2g3
    slow3!  -> f2g3
  }
  b2s3 {
    block2! -> b2s3
    free2!  -> f2s3
    stop3!  -> b2s3
    go3!    -> b2g3
    slow3!  -> b2g3
  }
  b2g3 { }
};

specification RqB2Y2L {
  states { f2n2, f2a2, b2n2, b2a2, e }
  inis   { f2n2 }
  errs   { e }
  channels { block2, free2, norm2, alt2 }
  f2n2 {
    block2! -> b2n2
    free2!  -> f2n2
    norm2!  -> f2n2
    alt2!   -> f2a2
  }
  f2a2 {
    block2! -> b2a2
    free2!  -> f2a2
    norm2!  -> f2n2
    alt2!   -> f2a2
  }
  b2n2 {
    block2! -> b2n2
    free2!  -> f2n2
    norm2!  -> b2n2
    alt2!   -> e
  }
  b2a2 {
    block2! -> b2a2
    free2!  -> f2a2
    norm2!  -> e
    alt2!   -> b2a2
  }
};

specification RqB3S0L {
  states { f3s0, f3g0, b3s0, b3g0 }
  inis   { f3s0 }
  errs   { b3g0 }
  channels { block3, free3, stop0, go0 }
  f3s0 {
    block3! -> b3s0
    free3!  -> f3s0
    stop0!  -> f3s0
    go0!    -> f3g0
  }
  f3g0 {
    block3! -> b3g0
    free3!  -> f3g0
    stop0!  -> f3s0
    go0!    -> f3g0
  }
  b3s0 {
```

```
    block3! -> b3s0
    free3!  -> f3s0
    stop0!  -> b3s0
    go0!    -> b3g0
  }
  b3g0 { }
};

specification RqB3S1L {
  states { f3s1, f3g1, b3s1, b3g1 }
  inis   { f3s1 }
  errs   { b3g1 }
  channels { block3, free3, stop1, go1 }
  f3s1 {
    block3! -> b3s1
    free3!  -> f3s1
    stop1!  -> f3s1
    go1!    -> f3g1
  }
  f3g1 {
    block3! -> b3g1
    free3!  -> f3g1
    stop1!  -> f3s1
    go1!    -> f3g1
  }
  b3s1 {
    block3! -> b3s1
    free3!  -> f3s1
    stop1!  -> b3s1
    go1!    -> b3g1
  }
  b3g1 { }
};

specification RqS0Y2L {
  states { s0n2, g0n2, s0a2, g0a2 }
  inis   { s0n2 }
  errs   { g0n2 }
  channels { stop0, go0, norm2, alt2 }
  s0n2 {
    stop0! -> s0n2
    go0!   -> g0n2
    norm2! -> s0n2
    alt2!  -> s0a2
  }
  g0n2 { }
  s0a2 {
    stop0! -> s0a2
    go0!   -> g0a2
    norm2! -> s0n2
    alt2!  -> s0a2
  }
  g0a2 {
    stop0! -> s0a2
    go0!   -> g0a2
    norm2! -> g0n2
    alt2!  -> g0a2
  }

};

specification RqS1Y2L {
  states { s1n2, g1n2, s1a2, g1a2 }
  inis   { s1n2 }
  errs   { g1a2 }
  channels { stop1, go1, norm2, alt2 }
  s1n2 {
    stop1! -> s1n2
    go1!   -> g1n2
    norm2! -> s1n2
    alt2!  -> s1a2
  }
  g1n2 {
    stop1! -> s1n2
    go1!   -> g1n2
    norm2! -> g1n2
    alt2!  -> g1a2
  }
  s1a2 {
    stop1! -> s1a2
    go1!   -> g1a2
    norm2! -> s1n2
    alt2!  -> s1a2
  }
  g1a2 {
  }
};

specification RqS3Y2L {
  states { s3n2, g3n2, s3a2, g3a2 }
  inis   { s3n2 }
  errs   { g3a2 }
  channels { stop3, go3, slow3,
    norm2, alt2 }
  s3n2 {
    stop3! -> s3n2
    go3!   -> g3n2
    slow3! -> s3n2
    norm2! -> s3n2
    alt2!  -> s3a2
  }
  g3n2 {
    stop3! -> s3n2
    go3!   -> g3n2
    slow3! -> s3n2
    norm2! -> g3n2
```

```
    alt2!  -> g3a2
  }
  s3a2 {
    stop3! -> s3a2
    go3!   -> g3a2
    slow3! -> s3a2
    norm2! -> s3n2
    alt2!  -> s3a2
  }
  g3a2 { }
};
specification Switch {
  states { n, a }
  inis { n }
  channels { norm, alt }
  n {
    norm?? -> { n }
    alt??  -> { a }
  }
  a {
    norm?? -> { n }
    alt??  -> { a }
  }
};

specification Block0 {
  states { f0, b0, e0 }
  inis { f0 }
  errs { e0 }
  channels { free0, block0 }
  f0 {
    free0?? -> f0
    block0?? -> b0
  }
  b0 {
    free0?? -> f0
    block0?? -> e0
  }
  e0 { }
};

specification Block1 {
  states { f1, b1, e1 }
  inis { f1 }
  errs { e1 }
  channels { free1, block1 }
  f1 {
    free1?? -> f1
    block1?? -> b1
  }
  b1 {
    free1?? -> f1
    block1?? -> e1
  }
  e1 { }
};

specification Block2 {
  states { f2, b2, e2 }
  inis { f2 }
  errs { e2 }
  channels { free2, block2 }
  f2 {
    free2?? -> f2
    block2?? -> b2
  }
  b2 {
    free2?? -> f2
    block2?? -> e2
  }
  e2 { }
};

specification Block3 {
  states { f3, b3, e3 }
  inis { f3 }
  errs { e3 }
  channels { free3, block3 }
  f3 {
    free3?? -> f3
    block3?? -> b3
  }
  b3 {
    free3?? -> f3
    block3?? -> e3
  }
  e3 { }
};
```

C.2.2. Composition of the System

The composition of the railway system is realised as a Go program that reads the component specifications and applies the composition operations parallel composition, conjunction and quotienting.

```
package main
```

```
import (
  "os"
  specParse "swt-bamberg.de/mia/specParser"
  mia "swt-bamberg.de/mia/miaDS"
  "log"
)

func main () {
  // Specifications
  s := make(map[string]mia.MiaAutomaton)
  p := specParse.NewParser()
  err := p.ParseFile(os.Stdin)

  if err != nil {
    panic ("parse error")
  }
  mc := p.GetResultMias()

  // Construct components
  for name, automaton := range mc {
    s[name] = automaton.Mia
  }

  // --- Small Example ------------------------------------------
  log.Println("Computing small example ...")
  s["pS0S1A"]   = mia.ParallelProductEmia(s["SignalRef3A0"], s["SignalRef3A1"])
  s["pS0S1B"]   = mia.ParallelProductEmia(s["SignalRef3B0"], s["SignalRef3B1"])
  s["qGR"]      = mia.Quotient(s["Global1"], s["RqS0S1"])
  s["qGSA"]     = mia.Quotient(s["Global1"], s["pS0S1A"])
  s["qGSB"]     = mia.Quotient(s["Global1"], s["pS0S1B"])
  s["ContrA"]   = mia.ConjunctionEmia(s["RqContr"], s["qGSA"])
  s["ContrB"]   = mia.ConjunctionEmia(s["qGR"], s["qGSB"])
  s["ContrObs"] = mia.ConjunctionEmia(s["qGR"], s["qGSA"])

  // --- Larger Example -----------------------------------------
  log.Println("Computing large example ...")

  // Track
  log.Println("* Track ...")
  s["Signals"]  = mia.ParallelProductEmia(s["pS0S1A"], s["Signal3"])
  s["Blocks01"] = mia.ParallelProductEmia(s["Block0"], s["Block1"])
  s["Blocks23"] = mia.ParallelProductEmia(s["Block2"], s["Block3"])
  s["Blocks"]   = mia.ParallelProductEmia(s["Blocks01"], s["Blocks23"])
  s["SB"]       = mia.ParallelProductEmia(s["Signals"], s["Blocks"])
  s["Track"]    = mia.RemoveUnreachable(
    mia.ParallelProductEmia(s["SB"], s["Switch"]))

  // Conjunction of R1, R2, R3
  log.Println("* Requirements R1-R3 ...")
  a1 := []string{"norm2", "alt2", "free0", "block0", "free1", "block1"}
  s["RqSignalsExt"] = mia.AlphabetExtensionEmia(s["RqSignalsL"], a1, mia.Output)
  a2 := []string{"stop0", "go0", "stop1", "go1", "free1", "block1"}
  s["RqS3Y2Ext"]    = mia.AlphabetExtensionEmia(s["RqS3Y2L"], a2, mia.Output)
```

```
a3 := []string{"stop0", "go0", "stop1", "go1"}
s["RqS3B1Y2Ext"] = mia.AlphabetExtensionEmia(s["RqS3B1Y2L"], a3, mia.Output)
s["cR1R2R3"] = mia.RemoveUnreachable(mia.ConjunctionEmia(
  mia.ConjunctionEmia(s["RqSignalsExt"], s["RqS3Y2Ext"]), s["RqS3B1Y2Ext"]))

// Conjunction of R4, R5
log.Println("* Requirements R4-R5 ...")
a4_0 := []string{"stop1", "go1", "stop3", "go3", "slow3", "norm2", "alt2"}
s["RqB2S0Ext"] = mia.AlphabetExtensionEmia(s["RqB2S0L"], a4_0, mia.Output)
a4_1 := []string{"stop0", "go0", "stop3", "go3", "slow3", "norm2", "alt2"}
s["RqB2S1Ext"] = mia.AlphabetExtensionEmia(s["RqB2S1L"], a4_1, mia.Output)
a4_3 := []string{"stop1", "go1", "stop0", "go0", "norm2", "alt2"}
s["RqB2S3Ext"] = mia.AlphabetExtensionEmia(s["RqB2S3L"], a4_3, mia.Output)
a5 := []string{"stop0", "go0", "stop1", "go1", "stop3", "go3", "slow3"}
s["RqB2Y2Ext"] = mia.AlphabetExtensionEmia(s["RqB2Y2L"], a5, mia.Output)
s["cR4R5"] = mia.RemoveUnreachable(mia.ConjunctionEmia(
   mia.ConjunctionEmia(s["RqB2S0Ext"], s["RqB2S1Ext"]),
   mia.ConjunctionEmia(s["RqB2S3Ext"], s["RqB2Y2Ext"])))

// Requirement R6
log.Println("* Requirement  R6 ...")
a6_0 := []string{"stop1", "go1"}
s["RqB3S0Ext"] = mia.AlphabetExtensionEmia(s["RqB3S0L"], a6_0, mia.Output)
a6_1 := []string{"stop0", "go0"}
s["RqB3S1Ext"] = mia.AlphabetExtensionEmia(s["RqB3S1L"], a6_1, mia.Output)
s["Rq6"] = mia.RemoveUnreachable(mia.ConjunctionEmia(s["RqB3S0Ext"],
   s["RqB3S1Ext"]))

// Conjunction of R7, R8, R9
log.Println("* Requirements R7-R9 ...")
a7 := []string{"stop1", "go1", "stop3", "go3", "slow3"}
s["RqS0Y2Ext"] = mia.AlphabetExtensionEmia(s["RqS0Y2L"], a7, mia.Output)
a8 := []string{"stop0", "go0", "stop3", "go3", "slow3"}
s["RqS1Y2Ext"] = mia.AlphabetExtensionEmia(s["RqS1Y2L"], a8, mia.Output)
a9 := []string{"stop1", "go1", "stop0", "go0"}
s["RqS3Y2Ext"] = mia.AlphabetExtensionEmia(s["RqS3Y2L"], a9, mia.Output)
s["cR7R8R9"] = mia.RemoveUnreachable(mia.ConjunctionEmia(
  mia.ConjunctionEmia(s["RqS0Y2Ext"], s["RqS1Y2Ext"]), s["RqS3Y2Ext"]))

// Conjunction of all requirements
log.Println("* Conjunction of all requirements ...")
acR1R2R3 := []string{"free2", "block2", "free3", "block3"}
s["cR1R2R3Ext"] = mia.AlphabetExtensionEmia(s["cR1R2R3"],acR1R2R3,mia.Output)
acR4R5 := []string{"free0", "block0", "free1", "block1", "free3", "block3"}
s["cR4R5Ext"]   = mia.AlphabetExtensionEmia(s["cR4R5"], acR4R5, mia.Output)
aRq6 := []string{"stop3", "go3", "slow3","free0", "block0", "free1", "block1",
  "free2", "block2", "norm2", "alt2"}
s["Rq6Ext"]     = mia.AlphabetExtensionEmia(s["Rq6"], aRq6, mia.Output)
acR7R8R9 := []string{"free0", "block0", "free1", "block1", "free2", "block2",
  "free3", "block3"}
s["cR7R8R9Ext"] = mia.AlphabetExtensionEmia(s["cR7R8R9"],acR7R8R9,mia.Output)
s["cR1-9"] = mia.RemoveUnreachable(mia.ConjunctionEmia(
  mia.ConjunctionEmia(s["cR1R2R3Ext"], s["cR4R5Ext"]),
```

```
    mia.ConjunctionEmia(s["Rq6Ext"], s["cR7R8R9Ext"]))

  // Global track requirement
  log.Println("* Quotient global/track ...")
  s["qGTrack"] = mia.Quotient(s["GlobalL"], s["Track"])

  // Overall controller requirement
  log.Println("* Overall controller requirement ...")
  s["ContrL"] = mia.RemoveUnreachable(
    mia.ConjunctionEmia(s["cR1-9"], s["qGTrack"]))

  // Output
  log.Println("Exporting to Graphviz ...")
  for name, automaton := range s {
    f, err := os.Create (name + ".dot")
    if err != nil {
      // TODO: error handling
    } else {
      f.WriteString(automaton.MiaAutomatonToDotString())
      f.Close()
    }
  }
  log.Println("Done.")
}
```

Bibliography

[Abr+91] Jean-Raymond Abrial, Matthew K. O. Lee, David Neilson, P. N. Scharbach, and Ib Holm Sørensen. "The B-Method". In: *VDM '91 - Formal Software Development, 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 21-25, 1991, Proceedings, Volume 2: Tutorials*. 1991, pp. 398–405. DOI: 10.1007/BFb0020001.

[Adl+06] B. Thomas Adler, Luca de Alfaro, Leandro D. da Silva, Marco Faella, Axel Legay, Vishwanath Raman, and Pritam Roy. "Ticc: A Tool for Interface Compatibility and Composition". In: *Computer Aided Verification (CAV)*. Vol. 4144. LNCS. Springer, 2006, pp. 59–62. DOI: 10.1007/11817963_8.

[AGN96] Samson Abramsky, Simon J. Gay, and Rajagopal Nagarajan. "Interaction categories and the foundations of typed concurrent programming". In: *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*. 1996, pp. 35–113.

[AH01a] Luca de Alfaro and Thomas A. Henzinger. "Interface Automata". In: *Foundations of Software Engineering (FSE)*. ESEC/FSE-9. Vienna, Austria: ACM, 2001, pp. 109–120. DOI: 10.1145/503209.503226.

[AH01b] Luca de Alfaro and Thomas A. Henzinger. "Interface Theories for Component-Based Design". In: *Embedded Software (EMSOFT)*. Vol. 2211. LNCS. Springer, 2001, pp. 148–165. DOI: 10.1007/3-540-45449-7_11.

[AH05] Luca de Alfaro and Thomas A. Henzinger. "Interface-based Design". In: *Engineering Theories of Software-Intensive Systems*. Vol. 195. NATO Science Series. Springer, 2005, pp. 83–104. DOI: 10.1007/1-4020-3532-2_3.

[Alf+05] Luca de Alfaro, Leandro D. da Silva, Marco Faella, Axel Legay, Pritam Roy, and Maria Sorea. "Sociable Interfaces". In: *Frontiers of Combining Systems (FroCoS)*. Vol. 3717. LNAI. Springer, 2005, pp. 81–105. DOI: 10.1007/11559306_5.

[And95] Henrik R. Andersen. "Partial Model Checking (Extended Abstract)". In: *IEEE Symposium on Logic in Computer Science*. IEEE, 1995, pp. 398–407. DOI: 10.1109/LICS.1995.523274.

[AP91] Luis M. Alonso and Ricardo Peña. "Acceptance Automata: A framework for specifying and verifying TCSP parallel systems". In: *Parallel Architectures and Languages Europe (PARLE)*. Vol. 506. LNCS. Springer, 1991, pp. 75–91. DOI: 10.1007/3-540-54152-7_59.

[Bau+10] Sebastian S. Bauer, Philip Mayer, Andreas Schroeder, and Rolf Hennicker. "On Weak Modal Compatibility, Refinement, and the MIO Workbench". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 6015. LNCS. Springer, 2010, pp. 175–189.

[Bau+12] Sebastian S. Bauer, Alexandre David, Rolf Hennicker, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. "Moving from Specifications to Contracts in Component-Based Design". In: *Fundamental Approaches to Software Engineering (FASE)*. Vol. 7212. LNCS. Springer, 2012, pp. 43–58. DOI: 10.1007/978-3-642-28872-2_3.

[BČK10] Nikola Beneš, Ivana Černa, and Jan Křetínský. *Disjunctive Modal Transition Systems and Generalized LTL Model Checking*. Tech. rep. FIMU-RS-2010-12. Faculty of Informatics, Masaryk University Brno, 2010.

[BČK11] Nikola Beneš, Ivana Černá, and Jan Křetínský. "Modal Transition Systems: Composition and LTL Model Checking". In: *Automated Technology for Verification and Analysis*. Vol. 6996. LNCS. Springer, 2011, pp. 228–242. DOI: 10.1007/978-3-642-24372-1_17.

[Ben+11a] Nikola Beneš, Jan Křetínský, Kim G. Larsen, Mikael H. Møller, and Jiří Srba. "Parametric Modal Transition Systems". In: *Automated Technology for Verification and Analysis (ATVA)*. Springer, 2011, pp. 275–289. DOI: 10.1007/978-3-642-24372-1_20.

[Ben+11b] Jesper Bengtson, Magnus Johansson, Joachim Parrow, and Björn Victor. "Psi-Calculi: A Framework for Mobile Processes with Nominal Data and Logic". In: *Logical Methods in Computer Science (LMCS)* 7.1:11 (2011), pp. 1–44. DOI: 10.2168/LMCS-7(1:11)2011.

[Ben+12] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas A. Henzinger, and Kim G. Larsen. *Contracts for System Design*. Tech. rep. 8147. INRIA, 2012.

[Ben+13] Nikola Beneš, Benoît Delahaye, Uli Fahrenberg, Jan Křetínský, and Axel Legay. "Hennessy-Milner Logic with Greatest Fixed Points as a Complete Behavioural Specification Theory". In: *Concurrency Theory (CONCUR)*. Vol. 8052. LNCS. Springer, 2013, pp. 76–90.

[Ben+15] Nikola Beneš, Jan Kretínský, Kim G. Larsen, Mikael H. Møller, Salomon Sickert, and Jirí Srba. "Refinement Checking on Parametric Modal Transition Systems". In: *Acta Informatica* 52.2 (2015), pp. 269–297. DOI: 10.1007/s00236-015-0215-4.

[Bey+07] Dirk Beyer, Arindam Chakrabarti, Thomas A. Henzinger, and Sanjit A. Seshia. "An Application of Web-Service Interfaces". In: *Web Services (ICWS)*. IEEE, 2007, pp. 831–838. DOI: 10.1109/icws.2007.32.

[BHW11] Sebastian S. Bauer, Rolf Hennicker, and Martin Wirsing. "Interface Theories for Concurrency and Data". In: *Theoretical Computer Science* 412.28 (2011), pp. 3101–3121. DOI: 10.1016/j.tcs.2011.04.007.

[BK82] Jan A. Bergstra and Jan W. Klop. *Fixed Point Semantics in Process Algebra*. Tech. rep. 208. Mathematical Centre Amsterdam, 1982.

[BML11] Sebastian S. Bauer, Philip Mayer, and Axel Legay. "MIO Workbench: A Tool for Compositional Design with Modal Input/Output Interfaces". In: *Automated Technology for Verification and Analysis (ATVA)*. Springer, 2011, pp. 418–421. DOI: 10.1007/978-3-642-24372-1_30.

[Bpel17] *BPEL – Web Services Business Process Execution Language Version 2.0*. 2017. URL: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (visited on 04/21/2017).

[Bpmn17] *BPMN – Business Process Model and Notation*. 2017. URL: http://www.bpmn.org/ (visited on 04/21/2017).

[BPS01] Johannes A. Bergstra, Alban Ponse, and Scott A. Smolka, eds. *Handbook of Process Algebra*. Amsterdam: Elsevier, 2001.

[BR08] Purandar Bhaduri and S. Ramesh. "Interface Synthesis and Protocol Conversion". In: *Formal Aspects of Component Software (FACS)* 20.2 (2008), pp. 205–224. DOI: 10.1007/s00165-007-0045-4.

[Buj+14] Ferenc Bujtor, Sascha Fendrich, Gerald Lüttgen, and Walter Vogler. *Nondeterministic Modal Interfaces*. Tech. rep. 2014-06. Institut für Informatik, Universität Augsburg, 2014.

[Buj+15] Ferenc Bujtor, Sascha Fendrich, Gerald Lüttgen, and Walter Vogler. "Nondeterministic Modal Interfaces". In: *Theory and Practice of Computer Science (SOFSEM)*. Vol. 8939. LNCS. Springer, 2015, pp. 152–163. DOI: 10.1007/978-3-662-46078-8_13.

[Buj+16] Ferenc Bujtor, Sascha Fendrich, Gerald Lüttgen, and Walter Vogler. "Nondeterministic Modal Interfaces". In: *Theoretical Computer Science* 642 (2016), pp. 24–53. DOI: 10.1016/j.tcs.2016.06.011.

[But00] Michael Butler. "csp2B: A Practical Approach to Combining CSP and B". In: *Formal Aspects of Computing* 12.3 (2000), pp. 182–198. DOI: 10.1007/PL00003930.

[BV14] Ferenc Bujtor and Walter Vogler. "Error-Pruning in Interface Automata". In: *Theory and Practice of Computer Science (SOFSEM)*. Vol. 8327. LNCS. Springer, 2014, pp. 162–173. DOI: 10.1007/978-3-319-04298-5_15.

[BV16] Ferenc Bujtor and Walter Vogler. "ACTL for Modal Interface Automata". In: *Application of Concurrency to System Design (ACSD)*. 2016, pp. 1–10. DOI: 10.1109/ACSD.2016.11.

[BY09] Andi Bejleri and Nobuko Yoshida. "Synchronous Multiparty Session Types". In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 241 (2009), pp. 3–33. DOI: 10.1016/j.entcs.2009.06.002.

[Cai11] Benoît Caillaud. *Mica: A Modal Interface Compositional Analysis Library*. 2011. URL: http://www.irisa.fr/s4/tools/mica/ (visited on 02/14/2017).

[Car+16] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. "Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types". In: *Concurrency Theory (CONCUR)*. Vol. 59. LIPIcs. Schloss Dagstuhl, 2016, 33:1–33:15. DOI: 10.4230/LIPIcs.CONCUR.2016.33.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Principles of Programming Languages (POPL)*. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[CDY14] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. "On the Preciseness of Subtyping in Session Types". In: *Principles and Practice of Declarative Programming (PPDP)*. ACM, 2014, pp. 135–146. DOI: 10.1145/2643135.2643138.

[CE81] Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logics of Programs*. Vol. 131. LNCS. Springer, 1981, pp. 52–71. DOI: 10.1007/BFb0025774.

[CG00] Luca Cardelli and Andrew D. Gordon. "Mobile Ambients". In: *Theoretical Computer Science* 240.1 (2000), pp. 177–213. DOI: 10.1016/S0304-3975(99)00231-5.

[CGH01] Andrea Corradini, Martin Große-Rhode, and Reiko Heckel. "A Coalgebraic Presentation of Structured Transition Systems". In: *Theoretical Computer Science* 260.1 (2001), pp. 27–55. DOI: 10.1016/S0304-3975(00)00121-3.

[Che+12] Taolue Chen, Chris J. Chilton, Bengt Jonsson, and Marta Z. Kwiatkowska. "A Compositional Specification Theory for Component Behaviours". In: *Programming Languages and Systems (ESOP)*. Vol. 7211. LNCS. Springer, 2012, pp. 148–168. DOI: 10.1007/978-3-642-28869-2_8.

[Chi13] Chris J. Chilton. "An Algebraic Theory of Componentised Interaction". PhD thesis. University of Oxford, 2013.

[CJK13] Chris Chilton, Bengt Jonsson, and Marta Kwiatkowska. *An Algebraic Theory of Interface Automata*. Tech. rep. RR-13-02. University of Oxford, 2013.

[CPT16] Luís Caires, Frank Pfenning, and Bernardo Toninho. "Linear Logic Propositions as Session Types". In: *Mathematical Structures in Computer Science* 26.3 (2016), pp. 367–423. DOI: 10.1017/S0960129514000218.

[CS08] Key One Chung and Jonathan D.H. Smith. "Weak Homomorphisms and Graph Coalgebras". In: *The Arabian Journal for Science and Engineering* 33.2C (2008), pp. 107–121.

[DG04] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Operating System Design and Implementation (OSDI)*. USENIX Association, 2004, pp. 137–150.

[Dil89] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT-Press, 1989.

[DIp+07] Nicolás D'Ippolito, Dario Fishbein, Howard Foster, and Sebastian Uchitel. "MTSA: Eclipse Support for Modal Transition Systems Construction, Analysis and Elaboration". In: *OOPSLA Workshop on Eclipse Technology eXchange*. ACM, 2007, pp. 6–10. DOI: 10.1145/1328279.1328281.

[DIp+08] Nicolás D'Ippolito, Dario Fishbein, Marsha Chechik, and Sebastian Uchitel. "MTSA: The Modal Transition System Analyser". In: *Automated Software Engineering (ASE)*. IEEE, 2008, pp. 475–476. DOI: 10.1109/ASE.2008.78.

[Ecl17] *The Eclipse IDE*. 2017. URL: https://www.eclipse.org/ide (visited on 03/03/2017).

[Fah+15] Uli Fahrenberg, Jan Křetínský, Axel Legay, and Louis-Marie Traonouez. "Compositionality for Quantitative Specifications". In: *Formal Aspects of Component Software (FACS)*. Springer, 2015, pp. 306–324. DOI: 10.1007/978-3-319-15317-9_19.

[Fen17] Sascha Fendrich. *Gemia: Modal Interface Automata in Haskell*. 2017. URL: https://github.com/sfendrich/gemia (visited on 02/02/2017).

[FG17] Sascha Fendrich and Johannes Gareis. "Comparison of Software-Tools Implementing Interface Theories". Unpublished notes. 2017.

[FL16a] Sascha Fendrich and Gerald Lüttgen. *A Generalised Theory of Interface Automata, Component Compatibility and Error*. Tech. rep. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik 98. Bamberg University, 2016.

[FL16b] Sascha Fendrich and Gerald Lüttgen. "A Generalised Theory of Interface Automata, Component Compatibility and Error". In: *Integrated Formal Methods (iFM)*. Vol. 9681. LNCS. Springer, 2016, pp. 160–175. DOI: 10.1007/978-3-319-33693-0_11.

[FS08] Harald Fecher and Heiko Schmidt. "Comparing Disjunctive Modal Transition Systems with an One-selecting Variant". In: *The Journal of Logic and Algebraic Programming* 77.1-2 (2008), pp. 20–39. DOI: 10.1016/j.jlap.2008.05.003.

[Gar15] Johannes Gareis. "Prototypical Integration of the Modal Interface Automata Theory in Google Go". MA thesis. Bamberg University, Germany, 2015.

[GB92] Joseph A. Goguen and Rod M. Burstall. "Institutions: Abstract Model Theory for Specification and Programming". In: *Journal of the ACM* 39.1 (1992), pp. 95–146. DOI: 10.1145/147508.147524.

[GH05] Simon J. Gay and Malcolm Hole. "Subtyping for Session Types in the Pi Calculus". In: *Acta Informatica* 42.2-3 (2005), pp. 191–225. DOI: 10.1007/s00236-005-0177-z.

[Gla01] Rob J. van Glabbeek. "The Linear Time – Branching Time Spectrum I". In: Elsevier, 2001. Chap. 1, pp. 3–99. DOI: 10.1016/B978-044482830-9/50019-9.

[Gla93] Rob J. van Glabbeek. "The Linear Time – Branching Time Spectrum II". In: *Concurrency Theory (CONCUR)*. Vol. 715. LNCS. Springer, 1993, pp. 66–81. DOI: 10.1007/3-540-57208-2_6.

[GM03] Dimitra Giannakopoulou and Jeff Magee. "Fluent Model Checking for Event-based Systems". In: *SIGSOFT Software Engineering Notes* 28.5 (2003), pp. 257–266. DOI: 10.1145/949952.940106.

[Go17] *The Go Programming Language*. 2017. URL: https://www.golang.org (visited on 02/02/2017).

[GR01] Roberto Gorrieri and Arend Rensink. "Action Refinement". In: *Handbook of Process Algebra*. Elsevier, 2001, pp. 1047–1147. DOI: 10.1016/B978-044482830-9/50034-5.

[GR09] Gregor Goessler and Jean-Baptiste Raclet. "Modal Contracts for Component-Based Design". In: *Software Engineering and Formal Methods (SEFM)*. IEEE, 2009, pp. 295–303. DOI: 10.1109/SEFM.2009.26.

[Gra17] *Graphviz – Graph Visualization Software*. 2017. URL: http://www.graphviz.org (visited on 02/02/2017).

[Grä79] George Grätzer. *Universal Algebra*. Springer, 1979. DOI: 10.1007/978-0-387-77487-9.

[Grä96] George Grätzer. *General Lattice Theory*. Birkhäuser, 1996. DOI: 10.1007/978-3-0348-9326-8.

[GW05] Jan F. Groote and Tim A.C. Willemse. "Parameterised boolean equation systems". In: *Theoretical Computer Science* 343.3 (2005), pp. 332–369. DOI: 10.1016/j.tcs.2005.06.016.

[GW99] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999. DOI: 10.1007/978-3-642-59830-2.

[Has17] *The Haskell Programming Language*. 2017. URL: https://www.haskell.org (visited on 02/02/2017).

[Hen85] Mathew Hennessy. "Acceptance Trees". In: *Journal of the ACM* 32.4 (1985), pp. 896–928. DOI: 10.1145/4221.4249.

[HIJ15] Lukáš Holík, Malte Isberner, and Bengt Jonsson. "Mediator Synthesis in a Component Algebra with Data". In: *Correct System Design*. Vol. 9360. LNCS. Springer, 2015, pp. 238–259. DOI: 10.1007/978-3-319-23506-6_16.

[HJ04] Jesse Hughes and Bart Jacobs. "Simulations in Coalgebra". In: *Theoretical Computer Science* 327.1 (2004), pp. 71–108. DOI: 10.1016/j.tcs.2004.07.022.

[HK15] Rolf Hennicker and Alexander Knapp. "Moving from Interface Theories to Assembly Theories". In: *Acta Informatica* 52.2–3 (2015), pp. 235–268. DOI: 10.1007/s00236-015-0220-7.

[HM80] Matthew Hennessy and Robin Milner. "On Observing Nondeterminism and Concurrency". In: *Automata, Languages and Programming (ICALP)*. Springer, 1980, pp. 299–309. DOI: 10.1007/3-540-10003-2_79.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hon93] Kohei Honda. "Types for Dyadic Interaction". In: *Concurrency Theory (CONCUR)*. Vol. 715. LNCS. Springer, 1993, pp. 509–523. DOI: 10.1007/3-540-57208-2_35.

[Hüt+16] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. "Foundations of Session Types and Behavioural Contracts". In: *ACM Comput. Surv.* 49.1 (2016), 3:1–3:36. DOI: 10.1145/2873052.

[HYC16] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty Asynchronous Session Types". In: *Journal of the ACM* 63.1 (2016), 9:1–9:67. DOI: 10.1145/2827695.

[KGG14] Dimitrios Kouzapas, Ramunas Gutkovas, and Simon J. Gay. "Session Types for Broadcasting". In: *Programming Language Approaches to Concurrency and Communication-Centric Software (PLACES)*. Vol. 155. EPTCS. Open Publishing Association, 2014, pp. 25–31. DOI: 10.4204/EPTCS.155.4.

[KNS92] Gerhard Keller, Markus Nüttgens, and August-Wilhelm Scheer. *Semantische Prozeßmodellierung auf der Grundlage "Ereignisgesteuerter Prozessketten (EPK)"*. Tech. rep. Institut für Wirtschaftsinformatik, Universität des Saarlandes, 1992.

[Koz83] Dexter Kozen. "Results on the propositional μ-calculus". In: *Theoretical Computer Science* 27.3 (1983), pp. 333–354. DOI: 10.1016/0304-3975(82)90125-6.

[KS13] Jan Křetínský and Salomon Sickert. "MoTraS: A Tool for Modal Transition Systems and Their Extensions". In: *Automated Technology for Verification and Analysis (ATVA)*. Vol. 8172. LNCS. Springer, 2013, pp. 487–491. DOI: 10.1007/978-3-319-02444-8_41.

[LAF06] Axel Legay, Luca de Alfaro, and Marco Faella. "An Introduction to the Tool Ticc". In: *Workshop on Trustworthy Software*. Vol. 3. OASIcs. Schloss Dagstuhl, 2006, pp. 1–32. DOI: 10.4230/OASIcs.TrustworthySW.2006.766.

[Lar89] Kim G. Larsen. "Modal Specifications". In: *Automatic Verification Methods for Finite State Systems*. Vol. 407. LNCS. Springer, 1989, pp. 232–246. DOI: 10.1007/3-540-52148-8_19.

[LL15] Marten Lohstroh and Edward A. Lee. "An Interface Theory for the Internet of Things". In: *Software Engineering and Formal Methods (SEFM)*. Vol. 9276. LNCS. Springer, 2015, pp. 20–34. DOI: 10.1007/978-3-319-22969-0_2.

[LML15] Lars Luthmann, Stephan Mennicke, and Malte Lochau. "Towards an I/O Conformance Testing Theory of Software Product Lines based on Modal Interface Automata". In: *Formal Methods and Analysis in SPL Engineering (FMSPLE)*. Vol. 182. EPTCS. Open Publishing Association, 2015, pp. 1–13. DOI: 10.4204/EPTCS.182.1.

[LML17] Lars Luthmann, Stephan Mennicke, and Malte Lochau. "Compositionality, Decompositionality and Refinement in Input/Output Conformance Testing". In: *Formal Aspects of Component Software (FACS)*. Vol. abs/1606.09035. Springer, 2017, pp. 54–72. DOI: 10.1007/978-3-319-57666-4_5.

[LNW07] Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski. "Modal I/O Automata for Interface and Product Line Theories". In: *Programming Languages and Systems (ESOP)*. Vol. 4421. LNCS. Springer, 2007, pp. 64–79. DOI: 10.1007/978-3-540-71316-6_6.

[LSW95] Kim G. Larsen, Bernhard Steffen, and Carsten Weise. "A Constraint Oriented Proof Methodology Based on Modal Transition Systems". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 1019. LNCS. Springer, 1995, pp. 17–40. DOI: 10.1007/3-540-60630-0_2.

[LT88] Kim G. Larsen and Bent Thomsen. "A modal process logic". In: *Logic in Computer Science (LICS)*. IEEE, 1988, pp. 203–210. DOI: 10.1109/LICS.1988.5119.

[LV06] Gerald Lüttgen and Walter Vogler. "Conjunction on Processes: Full-Abstraction via Ready-Tree Semantics". In: *Foundations of Software Science and Computation Structures (FOSSACS)*. Vol. 3921. LNCS. Springer, 2006, pp. 261–276. DOI: 10.1007/11690634_18.

[LV07] Gerald Lüttgen and Walter Vogler. "Conjunction on Processes: Full-Abstraction via Ready-Tree Semantics". In: *Theoretical Computer Science* 373.1–2 (2007), pp. 19–40. DOI: 10.1016/j.tcs.2006.10.022.

[LV10] Gerald Lüttgen and Walter Vogler. "Ready Simulation for Concurrency: It's Logical!" In: *Information and Computation* 208.7 (2010), pp. 845–867. DOI: 10.1016/j.ic.2010.02.001.

[LV11] Gerald Lüttgen and Walter Vogler. "Safe Reasoning with Logic LTS". In: *Theoretical Computer Science* 412.28 (2011), pp. 3337–3357. DOI: 10.1016/j.tcs.2011.03.015.

[LV12] Gerald Lüttgen and Walter Vogler. "Modal Interface Automata". In: *Theoretical Computer Science (TCS)*. Vol. 7604. LNCS. Springer, 2012, pp. 265–279. DOI: 10.1007/978-3-642-33475-7_19.

[LV13a] Gerald Lüttgen and Walter Vogler. "Modal Interface Automata". In: *Logical Methods in Computer Science (LMCS)* 9.3:4 (2013). DOI: 10.2168/LMCS-9(3:4)2013.

[LV13b] Gerald Lüttgen and Walter Vogler. "Richer Interface Automata with Optimistic and Pessimistic Compatibility". In: *Electronic Communications of the EASST (ECEASST)* 66 (2013).

[LVF15] Gerald Lüttgen, Walter Vogler, and Sascha Fendrich. "Richer Interface Automata with Optimistic and Pessimistic Compatibility". In: *Acta Informatica* 52.4–5 (2015), pp. 305–336. DOI: 10.1007/s00236-014-0211-0.

[LX90] Kim G. Larsen and Li Xinxin. "Equation Solving Using Modal Transition Systems". In: *Logic in Computer Science (LICS)*. IEEE, 1990, pp. 108–117. DOI: 10.1109/LICS.1990.113738.

[Mac71] Saunders MacLane. *Categories for the Working Mathematician*. Springer, 1971.

[Maz88] Antoni W. Mazurkiewicz. "Basic Notions of Trace Theory". In: *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*. Vol. 354. LNCS. Springer, 1988, pp. 285–363. DOI: 10.1007/BFb0013025.

[mCRL17] *micro Common Representation Language 2*. 2017. URL: https://www.mcrl2.org (visited on 04/19/2017).

[Mil71] Robin Milner. "An Algebraic Definition of Simulation Between Programs". In: *International Joint Conference on Artificial Intelligence*. IJCAI. Morgan Kaufmann, 1971, pp. 481–489.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Vol. 92. LNCS. Springer, 1980. DOI: 10.1007/3-540-10235-3.

[Mil89] Robin Milner. *Communication and concurrency*. Prentice Hall, 1989.

[MK99] Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programs*. Wiley, 1999.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. "A Calculus of Mobile Processes, Parts I and II". In: *Information and Computation* 100.1 (1992), pp. 1–77. DOI: 10.1016/0890-5401(92)90008-4.

[NB17] *The NetBeans Platform*. 2017. URL: https://www.netbeans.org (visited on 03/03/2017).

[Par81] David Park. "Concurrency and Automata on Infinite Sequences". In: *Theoretical Computer Science GI-Conference*. Vol. 104. LNCS. Springer, 1981, pp. 167–183. DOI: 10.1007/BFb0017309.

[Pet62] Carl Adam Petri. "Kommunikation mit Automaten". PhD thesis. Universität Hamburg, Germany, 1962.

[QS82] Jean-Pierre Queille and Joseph Sifakis. "Specification and verification of concurrent systems in CESAR". In: *International Symposium on Programming*. Vol. 137. LNCS. Springer, 1982, pp. 337–351. DOI: 10.1007/3-540-11494-7_22.

[Rac+11] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, and Roberto Passerone. "A Modal Interface Theory for Component-based Design". In: *Fundamenta Informaticae* 108.1-2 (2011), pp. 119–149. DOI: 10.3233/FI-2011-416.

[Rac08] Jean-Baptiste Raclet. "Residual for Component Specifications". In: *Formal Aspects of Component Software (FACS)*. Vol. 215. Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier, 2008, pp. 93–110. DOI: 10.1016/j.entcs.2008.06.023.

[Rut00] Jan J. M. M. Rutten. "Universal coalgebra: a theory of systems". In: *Theoretical Computer Science* 249.1 (2000), pp. 3–80. DOI: 10.1016/S0304-3975(00)00056-6.

[Sdl17] *SDL – Specification and Description Language*. 2017. URL: http://sdl-forum.org (visited on 04/22/2017).

[SH15] Antti Siirtola and Keijo Heljanko. "Parametrised Modal Interface Automata". In: *ACM Trans. Embedded Comput. Syst.* 14.4 (2015), 65:1–65:25. DOI: 10.1145/2776892.

[Som15] Fabio Somenzi. *CUDD: CU Decision Diagram Package*. 2015. URL: http://vlsi.colorado.edu/~fabio/CUDD/cudd.pdf (visited on 04/18/2017).

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual*. 2. Ed. Prentice Hall, 1992.

[STH17] Antti Siirtola, Stavros Tripakis, and Keijo Heljanko. "When Do We Not Need Complex Assume-Guarantee Rules?" In: *ACM Trans. Embed. Comput. Syst.* 16.2 (2017), 48:1–48:25. DOI: 10.1145/3012280.

[Uml17] *UML – Unified Modeling Language*. 2017. URL: http://www.uml.org/ (visited on 04/21/2017).

[WC01] Jim C. P. Woodcock and Ana L. C. Cavalcanti. "A Concurrent Language for Refinement". In: *IWFM'01: 5th Irish Workshop in Formal Methods*. BCS Electronic Workshops in Computing. 2001.

[Win87] Glynn Winskel. "Event structures". In: *Petri Nets: Applications and Relationships to Other Models of Concurrency*. Vol. 255. LNCS. Springer, 1987, pp. 325–392. DOI: 10.1007/3-540-17906-2_31.

[WN93] Glynn Winskel and Mogens Nielsen. "Models for Concurrency". In: *DAIMI Report Series* 22.463 (1993). DOI: 10.7146/dpb.v22i463.6936.

Index