COMPOSITIONAL VERIFICATION OF COMPONENT-BASED HETEROGENEOUS SYSTEMS

Yan Jin

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN THE SCHOOL OF COMPUTER SCIENCE

UNIVERSITY OF ADELAIDE

January 2004

Abstract

It is widely acknowledged that mathematically-based formal specification and verification methods can provide substantial help in revealing ambiguity and inconsistency of informal system descriptions, and increase confidence in the correctness of system designs. However, as their complexity rapidly grows, modern computer-based systems often consist of parts (or subsystems) with very different characteristics, *e.g.* data-centred vs. control-oriented. Traditional formal methods relying on a single specification language are no longer appropriate for coping with all aspects of such a system. A combination of languages to be used for system specification is required.

To meet this need, it is essential to devise sound principles and techniques for integrating various languages and for reasoning about the resultant component-based heterogeneous system. It is also essential to provide mechanical support for their formal verification. This thesis focuses on discrete-event systems and a class of specification languages, viz. graph-like visual languages. It takes a two-step approach to study these fundamental issues and seek practical, tool-supported solutions.

Firstly, an underlying methodological framework is proposed, which enables both the use of different languages (or formalisms) to describe different subsystems and the automatic verification of heterogeneous systems. This framework is built on a solid semantic base of interconnected discrete-event components. It uses a semantic interpretation approach to heterogeneous systems, specifying language-specific interpreters which enable heterogeneous components to be defined in terms of this semantic base. More specifically, the interpreters are parameterized with component models and execute on behalf of the components according to the semantics of the specification languages. They are also enhanced with facilities that allow analysis tools to access the state and transition information of the components and control their concurrent execution. As a consequence, not only is the exhaustive analysis of heterogeneous systems supported, but also an open and extensible platform is provided, which allows various graph-like languages and formal verification techniques to be used for specifying and reasoning about heterogeneous systems.

Secondly, this thesis focuses on model checking, a robust and largely automatic approach to system formal verification. It proposes a compositional approach to combat the state space explosion problem, a well-known obstacle to model checking. This approach divides a verification problem of a system into sub-problems of its components and then addresses each sub-problem independently. The key is to specify abstract communication protocols for components using a lightweight formal language — *interface automata* [54], and then utilise these protocols as behavioural contracts for independent analysis of the components and their composition patterns. It is demonstrated that, adopting this divide-and-conquer approach, not only the basic properties of component-based systems, such as consistency and deadlock freedom, but also their safety properties can be proved. Furthermore, this compositional approach is implemented as automated tools in the context of the Moses tool suite [65], utilising the semantic interpretation approach proposed earlier. These tools are then applied to the verification of a non-trivial distributed embedded system — the Production Cell [138]. It is shown that a significant reduction in complexity of system verification is achieved.

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

Yan Jin

January 2004

Acknowledgements

First and foremost, I am indebted to my supervisors, Dr. Robert Esser and Dr. Charles Lakos. I would never have accomplished this work without their constant support, guidance, and friendship.

It was Robert who introduced me to this research area. He has always been available and patient to answer my questions, and committed himself to offering his assistance when needed. His profound industrial experience and practical spirit has helped me many times to make the right choices; his enthusiasm and scholarly insights made the process of pursuing my PhD a joyful and fruitful journey.

It was Charles who spent a large amount of time reading and commenting on preliminary versions of this thesis and made numerous helpful and constructive suggestions for improving it. His spirited deep thinking made our discussions very stimulating; his scholarly insights were invaluable to my work.

I am grateful to Dr. Jörn W. Janneck at UC Berkeley for our exciting and fruitful collaboration. It was his work on which this thesis was partially built. I am also grateful to Prof. Monika Heiner at Brandenburg Technical University at Cottbus for her assistance in providing materials associated with her papers on the Production Cell.

I would like to express my sincere gratitude to Ping Gao and Bob Fang for their helpful assistance in both my work and life since my arrival in Adelaide. I would also like to thank Catherine Menon and Kylie Williams for creating a lovely, pleasant office atmosphere.

I wish to thank the University of Adelaide for the financial support for this research and its associated activities. I also wish to thank the School of Computer Science, especially the heads, Prof. Chris Barter and Prof. Mike Brooks, for providing an excellent research environment and the financial support for attending conferences.

Special thanks go to Lee and Keith Jurgs not only for their generous friendship but also for their warmhearted assistance in providing me with excellent opportunities to practise my English and become familiar with Australian culture.

Last but not least, I am deeply grateful to my wife, Lin Gong, for her love, care, understanding and sacrifice during this important period of my life. I am also grateful to my parents, Yongqing Xu and Weicai Jin, for their love, encouragement and support over the years and for shaping in me a character persistent enough to accomplish this work.

Contents

List of Figures

List of Tables

List of Programs

1
Introduction

1.1 Motivation

The rapid progress of computer technology has made an inevitable and dramatic increase in the scale and complexity of computer-based systems. As a result, there is an increased likelihood of subtle errors or defects creeping into system designs and remaining undetected. With computers becoming more pervasive, failures in computer-based systems have been more severe, sometimes resulting in catastrophic consequences such as the loss of money, time or even human lives.

This situation becomes more serious when it comes to concurrent systems. These systems are inherently more complex than their sequential counterparts, as they are composed of a number of components which communicate with each other and operate concurrently. In fact, the majority of today's large systems are concurrent. This makes it even more difficult to prevent computer failures.

It is therefore crucial to employ methodologies that can increase confidence in the correctness of system designs before they are actually implemented and used. It also makes economic sense to detect design defects as early as possible in the development process, since the later a defect is discovered, the harder and more expensive it is to fix [147].

Formal verification is a promising means for ensuring system correctness in early stages of the development process. It employs mathematically-based languages, techniques and tools for the specification, design and construction of computer-based systems. By means of logical and unambiguous specifications of system behaviour, it provides substantial help in revealing the ambiguity and inconsistency of informal system descriptions, and offers systematic approaches to system verification.

Model checking is one of the important approaches to the formal verification of concurrent systems. Given a system, this approach builds a finite representation of the design, called the *state space*, which enumerates all possible states of the system. It then formulates desired properties of the system and checks every state for the violation of these properties. Compared with other approaches such as theorem proving, model checking is largely automatic and easy to use. Furthermore, its ability to give counterexamples when a property is violated provides a very useful aid in locating and eliminating design defects. This approach is therefore very appealing to practitioners.

However, despite these advantages, model checking is, to a large extent, only popular in the research community and less frequently used in industry, especially for software development. Three major factors have limited its use by software engineers.

Firstly, model checking cannot be directly applied to infinite state systems. Even for finite state systems, it often suffers from the so-called *state space explosion* problem. That is, the size of state space of a system tends to grow exponentially with the number of its components and as a consequence quickly exceed the memory capacity of computers. This greatly limits the size of systems that can be handled by model checking. To attack this problem, a body of techniques have been proposed in the literature. These include symbolic methods, *e.g.* [148, 199], on-the-fly model checking, *e.g.* [70, 84], state space reduction, *e.g.* [32, 33, 75, 79, 105, 195], abstraction, *e.g.* [18, 35, 77], and compositional verification, *e.g.* [7, 41, 81, 92, 149]. Among them, compositional verification is a promising approach dividing a verification task on a system into sub-tasks on its components and then addressing each sub-task independently.

Secondly, it is often required in model checking approaches that a whole system is modelled using a single specification language. However, as modern complex systems often have aspects or parts with different characteristics, it may not be possible to find a single language suitable for specifying all parts. This motivates the employment of appropriate formalisms for describing different parts or aspects of systems. This approach, nevertheless, results in heterogeneous models and consequently the need to devise new formal techniques for their verification.

Thirdly, formal specification languages developed in the research community usually require a steep learning curve of software engineers, and the model checking methods recommended by researchers often require great effort to apply [90]. Engineers are often reluctant and sometimes unable to invest a large amount of time on learning such languages or methods. In order to be more accessible to engineers, methods based on languages that are easy to learn and use are in demand. For example, visual languages provide an intuitive means for studying design problems. Model checking methods using these languages for system specification seem to be more acceptable to engineers.

In Chapter 2, we shall present a more extensive investigation on these issues which further justifies these concerns.

1.2 Research Goal

The research reported in this thesis is aimed at addressing these concerns while providing formal verification methods or tools to practitioners. In particular, it builds on the previous work [65, 110] of the Moses project [2] on the modelling and simulation of component-based heterogeneous systems. It focuses on the formal verification of discrete-event systems and proposes lightweight and tool-supported approaches which can be easily integrated into the current practice of software development.

The major directions of this research are twofold. First of all, we need to develop an extensible formal verification framework, which

- allows complex systems to be specified using a combination of modelling languages. In particular, graph-like visual languages are investigated. These languages consist of vertices (represented as closed geometrical shapes) and edges (represented as lines) connecting vertices. Examples include Petri nets, statecharts and process networks;

- provides automatic tools for the verification of safety properties of heterogeneous systems;
- supports the easy accommodation of new languages and model checking techniques without requiring any changes to currently supported languages or techniques. The openness and extensibility of this framework is desirable because it is very likely that new notations and techniques are created with the continuous emergence of new application domains.

In addition, we need to develop new or employ currently available compositional verification techniques for the formal verification of heterogeneous systems, which

- make use of the component-based nature of these systems to alleviate the state space explosion problem;
- are lightweight, requiring of the user little knowledge of formal verification.

1.3 Contributions

This thesis makes a number of contributions towards the research goal. First of all, to support heterogeneous systems, we decouple the specification and verification processes, and relate them within a common semantic framework based on *discrete-event components* (DECs). As a reactive variant of labelled transition systems [13], DECs are sufficiently general to give semantics to many pragmatic discrete-event models. This approach brings three immediate benefits:

- The verification task of a heterogeneous system can be transformed to that of a homogeneous system of DECs, to which current model checking techniques can be applied;
- New model checking techniques can be easily added without requiring any changes to existing modelling languages;
- The designer will have the freedom to choose an appropriate language to specify each part of a heterogeneous system.

As pointed out by Day [52], decoupling is a key characteristic of a second generation approach to formal verification. Its usefulness will be demonstrated in this thesis by the independent development of modelling languages and model checking techniques.

We then adopt a component-based design approach based on DECs, and propose a novel compositional verification technique for the resultant systems. Here, we make a clear distinction between computational and compositional aspects of a system, letting DECs represent computationally intensive components and the interconnections between DECs represent component composition patterns. Correspondingly, the composition of DECs is called *DEC networks*. To enable the compositional verification of DEC networks, we make explicit the behavioural contracts between components and their composition, and employ interface automata (IAs), a lightweight formalism proposed by de Alfaro and Henzinger [54], to formally describe them. Such a contract covers the services a component provides, the way it reacts to its inputs and what it expects from its environment. This, however, does not disclose implementation detail of the component. Hence a contract is often referred to as the abstract *communication protocol* that a component must conform to in the system (or network). With the contracts described at a sufficient level of abstraction, we can then utilise them for independent analysis of both components and their composition patterns. As a result, the state space explosion problem in the verification of DEC networks can be alleviated.

Furthermore, we rely on the semantic framework of DECs and adopt a practical approach to supporting the use of multiple formalisms for system specification. More specifically, we extend the previous work of the Moses tool suite [65] on the semantic interpretation of graph-like languages, and propose a semantic interpretation approach to heterogeneous systems, which enables their exhaustive state space analysis. In this approach, language-specific interpreters are defined and, when parameterized by component models, execute on behalf of the components according to the semantics of the modelling languages. The interpreters are also enhanced with facilities that allow analysis tools to access the state and transition information of the components and control their execution. As a consequence, a formal semantics of heterogeneous systems can be given in terms of DECs, provided analysis tools are able to coordinate the concurrent execution of components according to their interconnections. In this way, we make it possible to apply sophisticated model checking techniques developed for DECs or more general transition systems, including the above-mentioned compositional approach, to the verification of heterogeneous systems. In addition, it is worth noting that this interpretation approach is generic to the languages under consideration of this thesis, *i.e.* graph-like languages. Accordingly, an extensible verification framework for heterogeneous systems can be obtained.

Based on the proposed interpretation approach, we discuss the implementation of the above-mentioned compositional verification approach in the context of the Moses tool suite. This includes the incorporation of the formalism of interface automata and the tool development for IA composition, IA compatibility checking and the independent analysis of heterogeneous components (*i.e.* components coded in various modelling languages).

Finally, we apply the proposed compositional approach and the developed tools to the verification of a non-trivial distributed embedded system — the Production Cell [138]. In this case study, we show a significant reduction on the size of the state space that needs to be built for verification.

1.4 Thesis Outline

The remainder of this thesis is organised as follows. In Chapter 2, the background of this work is outlined. This includes an extensive investigation of formal verification techniques, multi-formalism modelling approaches, component-based design practice as well as modelling languages.

In Chapter 3, the foundation for the above-mentioned compositional verification approach is laid. This includes

- definitions of discrete-event components, interface automata and their composition;
- formulations of conformance relations between DECs and IAs, together with practical conformance checking methods;
- formulations of the consistency, deadlock freedom and safety properties for DEC networks;

- presentations of compositional verification methods for both closed and open DEC networks as well as the theoretical justifications.

In addition, comparisons of this approach with other existing compositional verification approaches are made.

In Chapter 4, a semantic interpretation approach to heterogeneous systems is presented. This imposes a behavioural contract between analysis tools and language-specific component interpreters so as to enable the exhaustive state space exploration of heterogeneous systems. Languages such as Petri nets and UML statecharts are used for demonstration of interpreter specification. As such, the relation with the previous work is also described.

In Chapter 5, two verification approaches to component-based heterogeneous systems are implemented, utilising the interpreter facilities developed in Chapter 4. These include a monolithic approach and the compositional approach proposed in Chapter 3. The implementation of the monolithic approach involves a simple algorithm for constructing the system state space and solutions to the specification and verification of safety properties. The implementation of the compositional approach builds on the above and includes algorithms for checking the conformance of heterogeneous components with IAs, for checking the compatibility of IAs, and for compositionally verifying safety properties.

In Chapter 6, a case study on the Production Cell is presented. This includes its design and verification methodologies as well as the experimental results with our compositional verification approach. It is shown that not only basic properties such as consistency and deadlock freedom but also important safety properties can be verified compositionally. It is also shown that approximately three orders of magnitude improvement in the size of the required system state space was achieved.

Finally in Chapter 7, a summary of our contributions is presented together with proposals for future work.

1.5 Relation to Previous Publications

Some parts of this thesis have been published or accepted for publication. Chapter 3 constitutes a more detailed presentation of [120, 121, 123]. It reformulates that work in the regular framework of Arnold and Nivat [13]; it uses the notion of communication ports for passing information between processes (or components); it takes a more realistic approach of constraining the type of messages that are transferred via each port; and it extends that work with solutions to the compositional verification of safety properties.

Chapter 4 integrates the work of [117, 118, 119] in the context of formal verification. It incorporates an explicitly and more clearly defined contract between analysis tools and component interpreters; It takes a more realistic approach to interpreter specification; It extends the work of [117] on the semantic definition of UML statecharts, with a refined strategy of solving transition conflicts and a more complete solution to handling completion events.

Finally, Chapter 6 and Appendix A contain a more detailed presentation of [122].

2 Background

In this chapter, we shall outline the background of this research from four perspectives: formal verification, multi-formalism modelling, component-based design, and modelling languages.

2.1 Formal Verification

Today, the rapidly growing complexity of computer-based systems brings the potential for a higher number of errors in their designs. Detecting and eliminating errors as early as possible is an economic imperative, since it is harder and more expensive to fix them in later development stages [147]. Furthermore, safety-critical systems, such as air traffic control, railway signalling, medical instruments and electronic commerce, require the absence of errors in their designs, as failures in these systems can result in disastrous consequences [39]. All the above indicate a pressing need for ensuring the correctness of systems at early stages of the design cycle.

Traditionally, two validation methods are exploited to reduce errors: *simulation* and *testing*. Both of them validate a system by feeding it with well-chosen input samples and evaluating the outputs. Simulation is based on a design model, while testing is on an actual product. When an error is uncovered, debugging is performed in order to find the cause and eliminate the error. These methods are effective in detecting errors. However, their effectiveness drops quickly, as the design or product becomes cleaner with less defects [39]. As a consequence, one can never be sure that all errors have been removed from the design or even how many defects may still be at large. In other words, simulation or testing fails to guarantee the freedom of a system from errors.

Formal verification is a promising alternative to simulation and testing. In contrast to the latter, formal verification builds on mathematically-defined system models and requirements (or *desired properties*), and conducts an exhaustive exploration of all possible behaviours of a system model against its requirements. It is thus able to thoroughly evaluate the system behaviours and ultimately ensure the correctness of the system against the requirements. It also provides practising engineers with a solid methodological framework allowing them to combine their experience with mathematical rigour. Experience has shown that even partial application of formal verification in the system development can lead to a significant quality increase and a reduction of the total development cost [175, 184].

In formal verification, two classes of system requirements are usually distinguished: *safety properties*, which require that bad things never happen, and *liveness properties*, which demand that good things will eventually happen.

Furthermore, there are two well-established approaches to formal verification: theorem proving and model checking [43]. They differ in the way of describing system models and proving system requirements. In the subsequent sections, details about these approaches are given.

2.1.1 Theorem Proving

Theorem proving methods describe both a system and its desired properties by assertions or formulae in some mathematical logic, and use axioms and inference rules to prove the satisfaction of the properties. These methods have been successfully applied to solve various combinatorial problems [43]. The main strength of these methods lies on the fact that they are generally applicable and can directly handle systems with infinite state spaces. They can also verify stronger properties than model checking methods [178]. Popular theorem proving tools include ACL2 [125], HOL [76], and PVS [161].

Though acknowledged as a powerful technique, theorem proving methods have some weaknesses that prevent them from being widely used among software engineers. First of all, the proving process is usually outside the scope of fully automated procedures and requires manual guidance [199]. It is prone to human errors and notoriously time-consuming [136]. Further, theorem proving methods seldom provide a counterexample that might help the user analyse the reason and locate the error when they fail to prove the correctness of a system [178]. Additionally, the employment of these methods requires of the user an expertise in advanced mathematics and clever proof strategies [90]. That is to say, this requires a strong understanding of operations of a proving tool in order to successfully guide it and diagnose when it fails [178]. All the above hinder the full automation of theorem proving methods and largely limit their application in industry.

2.1.2 Model Checking

Model checking methods [37] are aimed at automatic analysis and verification of concurrent systems with a finite number of states. For a given system, they rely on constructing a finite representation of the system, known as the *state space*, which consists of all states that the system can reach and all transitions that it can make between those states. Typically, the state space is represented in a mathematical structure such as a state transition system and an automaton. The construction of the state space is usually automatic.

Meanwhile, these methods formally express system properties in a mathematical logic (e.g. Temporal Logic [36]) or an automaton. They then perform the verification task of the system as an exhaustive exploration in its state space to make sure that the properties hold at every exploration step. The exploration is guaranteed to terminate since the model is finite. In some approaches such as [177], system properties are described as a finite model at a higher level of abstraction than the system model using the same description language. The verification (or model checking) task is then converted into checking the refinement between these two models.

In contrast to theorem proving, model checking is largely automatic and fast in most cases, and can be used by less trained personnel. Also, its ability to provide counterexamples is a useful aid in debugging and uncovering subtle errors. There are a large number of automatic model checking tools available both in the research community and in industry, e.g. SMV [148], SPIN [96], Murϕ [58], STeP [21], HyTech [11], MOCHA [12], Concurrency Workbench [44], FDR [177], Design/CPN [1], Maria [145], CADP [66], and

UPPAAL [4]. All these make model checking attractive for designers working in a wide range of application domains.

Certainly, model checking methods also have disadvantages. Above all, they cannot be directly applied to infinite state systems since the state exploration is thus unable to terminate. In addition, with systems being more complicated, these methods easily run into the *state space explosion* problem. This severely limits the size of systems that can be handled.

2.1.3 Methods for Attacking the State Space Explosion

The benefits promised by model checking have motivated researchers to address its limitations. A large number of proposals have been suggested in the literature to alleviate the state space explosion problem. The proposed methods are able to reduce the number of states that need to be stored for verifying certain kinds of properties, and thus considerably increase the size of systems that can be verified. Roughly, these methods can be classified into the following categories: optimised data structures, on-the-fly model checking, state space reduction, abstraction, and compositional verification.

Symbolic methods such as [100, 148, 163, 199] use concise data structures to represent state transition systems implicitly. These structures include *Binary Decision Diagrams* (BDDs) [27] and *Multi-valued Decision Diagrams* (MDDs) [124]. Symbolic methods succinctly encode sets of states and transitions as well as the state space as directed acyclic graphs which are compact and canonical for a given ordering of the input variables. Basic set operations including intersection, union and equivalence are performed efficiently with the BDD or MDD representations [128]. As a result, exploration and checking algorithms that utilise these operations can be executed efficiently. Experience has shown that symbolic model checking methods are particularly effective for systems with regular structure such as hardware circuits [28]. At the same time, these methods also face a real challenge. The size of BDDs or MDDs greatly depends on the ordering of the input variables, whereas finding the best ordering which results in a minimal BDD or MDD representation is NP-complete [27]. Hence manual efforts for investigating system structures and choosing an appropriate ordering are often needed.

On-the-fly methods verify a system without storing the whole state space. They are based on the observation that for verifying a certain class of properties, it is adequate to visit (rather than store) all the states and transitions. A typical on-the-fly method performs a depth-first exploration on the state space, and at each step of the exploration it stores only the current path from the initial state(s). As a result, the memory requirements for verifying these properties are substantially reduced [74]. However, the time needed to perform the exploration may grow dramatically due to the regeneration of already-visited states. Accordingly, other on-the-fly methods such as [16, 70, 84] attempt to offer reasonable compromises between time and memory requirements. On the whole, an important advantage of on-the-fly methods is that they can immediately generate a counterexample and stop the state space exploration once an error is uncovered. They are thus of substantial value in detecting errors at early stages of design when many errors tend to exist.

State space reduction relies on transforming a given verification problem into an equivalent problem on a smaller state space. This category further includes partial order reduction, symmetry reduction, garbage reduction and compositional minimisation.

Partial order reduction is based on the observation that in concurrent systems, the correctness of some properties is independent of the order of interleaved events. A selective search on the system state space is adequate to determine whether a property holds or not. As a result, the generation of all possible interleaving paths can be avoided. Notable approaches to partial order reduction are *stubborn sets* [195], *persistent sets* [73, 75], and *ample sets* [165].

Furthermore, the basic idea of *symmetry reduction* is to make use of symmetry bijections existing in many systems and to store only one representative state for a set of reachable states [196]. Consequently, the number of states that need to be stored is reduced but full information on the reachable state space is preserved. Notable approaches to symmetry reduction are [38, 62, 105, 115].

Additionally, *garbage reduction* methods attempt to delete or throw away information about encountered states, while constructing the state space. Typically, the states that are not reachable from the unexplored states are removed from the state space. As a result, a reduction on memory usage for storing the state space can be achieved. Methods falling into this category include *sweep-line* [33, 127], *bitstate hashing* [95, 97], *state space caching* [114], *pseudo-root states detecting* [164, 163], etc.

Finally, *compositional minimisation* (or *compositional reachability analysis*) is based on intermediate minimisation of subsystems (or components) using some semantic equivalence that preserves the property to be checked. It makes use of the hierarchical structure of a system and performs minimisation in steps, from the lowest to the highest level of the hierarchy. At each step, it minimises the state space of a subsystem by collapsing semantically equivalent states to a single state such that the properties to be checked are preserved. Ultimately, a minimised global state space is obtained which still contains enough information to verify the property under concern. However, as often the contextual constraints are not considered when subsystems are minimised, their state spaces may explode faster than the system [71]. This problem is called the *intermediate state space explosion*. To address it, many proposals have been suggested. Notable examples include [32, 71, 78, 79, 193, 194, 200]. Typically, additional interface processes capturing the contextual constraints are composed with subsystems, in order to suppress the execution sequences that never happen in the system and ultimately control the size of intermediate state spaces.

Abstraction methods focus on data values in concurrent systems that involve data paths [39]. These methods attempt to reduce the size of the state space to be handled by abstracting away the variables or data values irrelevant to a property to be checked. The abstraction is based on the observation that simple relationships usually exist between data values in the system specifications. Typically, a mapping function is built which maps the actual data values into a small set of abstract data values and consequently transforms a concrete system model into a (simpler) abstract model such that the satisfaction of a property on the abstract model implies that on the concrete model. The key factor of this approach is to select a good mapping function, because the abstract model has to be small and simple to facilitate a full state space exploration but still contain enough information for verifying the property under concern [142]. Many proposals to the construction of such a function have been presented in the literature, *e.g.* [17, 18, 38, 45, 49, 50, 51, 77, 99, 129, 133, 179, 180].

Compositional verification (or *compositional reasoning*) such as [130, 131, 197] attempts to combat the state space explosion using the principle of "divide and conquer", based on the fact that many systems are composed of multiple processes (or components) running in parallel. It divides the verification problem of a complex system into sub-problems of its components and addresses each sub-problem independently. The motivation behind this is that, to verify some properties, it is sufficient to consider only a subset of the components. Basically, in this approach, a property to be checked is first decomposed into local properties on some components. Then each local property is checked independently on the corresponding component, and the results are utilised to deduce the system property. Clearly, this approach does not require the construction of the global state space. Accordingly, the state space explosion problem can be alleviated.

At the same time, this approach also faces a challenge that often local properties of components are satisfied only when certain assumptions are met on their environment. That is to say, model-checking components in isolation sometimes yields a negative answer.

These are two solutions to this problem. The first solution was proposed by Clarke *et al.* [41]. It models the contextual constraints for each component as an abstract interface process and then uses a rule of inference, called the *interface rule*, to ensure the preservation of local properties of components at the system. Normally, an interface process can be derived from the other components in the system. The work adopting this solution includes [41, 42].

The second solution was proposed by Pnueli [171]. It specifies each component in terms of the properties that it assumes about its environment, and the properties that it would then guarantee, provided the assumptions hold. Thus, composing a component c with another component which (unconditionally) guarantees the assumptions of c will produce a system where the guarantees of c are enforced. Accordingly, this style of verification is often called *assume-guarantee reasoning*. The work along this line includes [81, 143, 146, 171, 188].

A limitation of this assume-guarantee reasoning paradigm is that it cannot directly handle systems involving interdependent assumptions between components. More specifically, suppose a system consisting of two components c and d such that it is necessary assume the correctness of d to verify c and vice versa. Then a circle would be encountered for verifying either c or d, and there is no place to begin the assume-guarantee chain. To break the circularity, a number of approaches to *circular reasoning* have been presented, such as [6, 7, 10, 92, 93, 128, 149, 150, 151]. They impose certain restrictions on the components and the properties to be checked such that the assume-guarantee inference rule remains valid by induction.

In general, it is a complicated task for compositional verification to decompose global properties of a system into local properties of the components. Also, it must be ensured that the satisfaction of local properties by the components implies the satisfaction of some global properties by the system. Furthermore, lightweight techniques and automated supporting tools are needed to make compositional verification widely acceptable by software engineers.

Admittedly, no single formal verification technique is able to solve all classes of problems. The combination of various techniques with tool support is therefore a worthwhile endeavor since it provides more powerful means to handle real problems. This has been demonstrated by some earlier work such as [21, 66, 96, 160].

2.2 Multi-Formalism Modelling

As mentioned above, formal verification is a mathematically-based methodology for describing and reasoning about the behaviour of a system. Traditionally, the designer uses a single formal language to describe a whole system for verification. However, with their complexity rapidly growing, modern computer-based systems often incorporate very different aspects or characteristics. In many cases, no single conceptual model or formalism is adequate for coping with all aspects of a system [108, 155, 141]. For example, a software system may be composed of data-centred and control-oriented components. Data-centred components can be naturally depicted using data-flow diagrams, whereas state-based diagrams are best suited for describing control-oriented components. Modelling such a system using a single language seems to be cumbersome and sometimes even infeasible with the resulting specification becoming too complicated. This obliges the designer to use appropriate formalisms, techniques and tools to deal with different parts or aspects of a system.

Furthermore, the specification of a complex system often involves a collaboration of groups of people from different disciplines. Each group tends to use their preferred methods or techniques to deal with the system. For example, domain experts prefer domain-specific languages, not only because they are familiar with these languages but also because these languages are usually lightweight and contains necessary domain-specific constraints which simplify the modelling task. Practical formal methods need to accommodate such heterogeneity.

Therefore, what is needed are various formalisms with ability to cope with specific problem domains, as well as the combination and integration of these formalisms in specifying the same system. The combination and integration can take advantage of individual formalisms in helping to generate more concise and understandable system specifications. Many modern software engineering methods, such as the Specification and Description Language (SDL) [106], the Unified Modelling Language (UML) [159], the Requirements State Machine Language (RSML) [137], the Software Cost Reduction language (SCR) [91], Rhapsody[85] and ROOM [182], recommend that the different aspects of a system be specified by different languages.

A significant amount of research has been carried out in the area of multi-formalism modelling (or heterogeneous specification). Depending on their focuses, existing approaches can be roughly classified into three categories: heterogeneous specification on components, on aspects, and on viewpoints.

Approaches to heterogeneous component specification use suitable languages to specify each part (or subsystem) of a system so as to more accurately and naturally capture the diverse characteristics present in these parts. Notable approaches include CodeSign [64], MOOSE [48], DOME [98], GME [102], Ptolemy II [19, 61], Moses [65], etc. The ultimate systems, often called *component-based heterogeneous systems*, or *heterogeneous systems* for short, are the main subject studied in this thesis.

Approaches to heterogeneous aspect specification use different languages to describe each aspect of a system. These include [53, 91, 137, 159, 181]. For example, the

UML [159] distinguishes between static and dynamic aspects of a system. It uses class diagrams to capture the static relation such as inheritance and aggregation between objects in a system, and statechart diagrams to describe the dynamic behaviour of the objects. The work of [181] takes a similar approach but is more flexible, allowing the static/dynamic aspects to be described by a variety of languages. Furthermore, the work of [53] distinguishes aspects on model, action, event and expression. It considers models to govern the state-transition aspect of systems, while using actions, events and expressions to label transitions between states. As above, this approach supports the specification of each aspect using various formalisms.

Approaches to heterogeneous viewpoint specification use different languages to describe the dynamic behaviour of a system from different perspectives. A perspective represents a view of the system in a particular domain. For example, in the UML, system behaviour can be described by use cases from the observer's point of view, by statecharts from objects' perspectives, and by sequence diagrams or collaboration diagrams with a view to the interactions between objects. Accordingly, approaches in this category are often called *multi-viewpoint approaches*. Other notable multi-viewpoint approaches include Catalysis [60], Rosetta [8], Viewpoint Oriented Software Development [67], the work of [22], etc. A distinct characteristic of these approaches is that behavioural models of different viewpoints often have some form of overlap [185]. Consequently, ensuring the consistency between these models is very important. Along this line, many proposals have been made, for example, [68, 69, 157, 191] to name a few.

Note that the above classification is not strict, since many systems or approaches exhibit heterogeneity in artifacts of multiple categories. For example, the UML consists of notations which can be used to describe a system on different aspects and from different viewpoints. As we were interested in systems composed of components with different characteristics, in this thesis we focus on component-based heterogeneous systems, as described in the first category.

2.2.1 Semantic Approaches to Heterogeneous Systems

In order to employ multiple formalisms for system specification, it is important to develop a solid semantic base, on which the composition of instances of the formalisms can be given an unambiguous and consistent semantics. Furthermore, in order to avoid the development of different techniques or the use of separate tools for each formalism, it is also important to construct an underlying framework, preferably with tool support, which builds on the semantic base and seamlessly unifies these formalisms. Only when these foundations are laid does the formal verification of the resultant heterogeneous system designs become possible.

To develop these foundations, it is essential to focus on a class of formalisms that is sufficiently narrow so that the common features can be exploited for developing sound principles for reasoning about heterogeneous systems and providing mechanical support for formal verification [167]. For example, [110] focuses on visual languages that can represent discrete-event systems and studies issues about their syntax and semantics definition for the modelling and simulation of heterogeneous systems. Similarly, [167] focuses on state-transition models and present an approach to the construction of analysis tools for heterogeneous systems made up of these models.

In the literature, a number of semantics approaches to heterogeneous system specifications have been presented. According to their employed methodologies, these approaches can be classified into three categories: syntactical translation between formalisms, syntactical translation into one common formalism, and semantic interpretation into a common semantic base.

Approaches to syntactical translation between formalisms such as [55, 162] first select a destination formalism for a given heterogeneous system and then translate all the component models written in other languages into it. Thus the semantics of the heterogeneous system is given in terms of the resultant model in the destination formalism. The verification of the system can then be supported by tools developed for the destination formalism.

Undoubtedly, these approaches are conceptually simple and able to utilise the tools developed for different languages. However, they require bidirectional translation between any pair of formalisms. This means that they need to restrict features of the supported languages to those that are translatable between all the languages. In other words, only an intersection of these languages can be accommodated. As admitted by Paige [162], even though using multiple languages for specification, one can express nothing more than just using one of them. Also, it is clear that the number of translations grows quadratically with the number of the supported languages. This implies that the addition of a new language requires a considerable effort to build bidirectional translations with all existing languages.

Approaches to syntactical translation into one common formalism select one common formalism as the destination language for translation, and specify the semantics of a given heterogeneous system in terms of the common formalism. For example, the work reported in [202, 203] employs one-sorted first-order predicate logic as the common formalism and supports the use of a combination of Z language [186], first-order logic, and finite automata for system specification. The composition of components (or partial specifications) of a system is equivalent to the conjunction of the assertions translated from them. In this way, this approach provides a very useful means for reasoning about heterogeneous systems.

In some approaches such as [30, 14], the input language of an existing analysis tool is chosen as the destination language in order to utilise the tool for formal verification. In this way, these approaches bridge the gap between notations developed for readability and understandability and notations developed for verification. They thus avoid the need to build formal analysis tools for each supported language and can take advantage of the advanced techniques already implemented in the analysis tool.

However, these approaches may have to leave out special features of all involved modelling languages, or extend the common formalism to represent these features. For instance, it is often the case that some data types of a modelling language are not supported by or are incompatible with the data types of the common formalism. Then an alternative solution has to be sought. Accordingly, the addition of a new language may require a slimming-down of the language or an extension of the common formalism to accommodate special features of the language.

Furthermore, in modelling heterogeneous systems, different formalisms are used because of their appropriateness for representing special characteristics of individual components (or subsystems). In other words, the use of a lot more constructs in the common formalism may be required to express some constructs in a modelling language which are specialised and optimized to a particular domain. This implies that a single common formalism may be cumbersome or unsatisfactory to represent some components [167], and syntactical translation may result in loss of information or produce a very complicated model in the common formalism.

Approaches to semantic interpretation into a common semantic base such as [110, 166, 167, 59, 189, 156] differ from the second category in that they usually build on a general semantic base (*e.g.* state transition systems) and focus on the aspects of individual modelling languages that are essential for defining the operational semantics of heterogeneous systems in terms of the semantic base (*e.g.* in terms of states and transitions). These approaches usually do not directly deal with the syntactic issues, such as constructs, data types, expressions and special syntactical features, or construct substitutability between languages. Consequently, simpler solutions to the semantic definition of heterogeneous systems are obtained.

For instance, [110] focuses on the language aspects that attribute to states and transitions between states, and defines language-specific interpreters using Abstract State Machines (ASMs) [112], which characterise the semantics of heterogeneous components in terms of dataflow actors. As a result, the semantics of heterogeneous systems is given in terms of networks of dataflow actors. Furthermore, [167] identifies a simple common structure, called *labelled directed hypergraphs*, to capture the essential information for the semantics definition of heterogeneous components. It then specifies a set of rules for each language, which operate on hypergraphs and render the state and transition information of heterogeneous components. As a result, heterogeneous components are interpreted as state transition systems and heterogeneous systems as the composition of transition systems.

In summary, to define the semantics of heterogeneous systems, semantic interpretation approaches do not require the syntactical translation between individual formalisms and thus avoid the problems faced by the translation approaches. Also, the use of a common semantic base largely simplifies the interpretation of heterogeneous components and their composition. Therefore, in this thesis, to support the verification of heterogeneous systems, a semantics interpretation approach, in particular, the approach proposed by [110], is taken. This will be detailed in Chapter 4.

2.3 Component-Based Design

In recent years, component-based design has become an important approach to building complex systems. It is concerned with designing components as reusable units, designing systems by reusing components, and maintaining/upgrading system designs by means of component replacement or customization [47]. In contrast to classical top-down design, the main idea of component-based design is to reuse existing solutions to well-studied problems in terms of components. In this way, a promising means of achieving software reuse, rapid development and complexity management is provided. Throughout this thesis, we shall call the systems adopting such a design methodology as *component-based systems* (CBS).

Although component-based design has been a widely used term, there exists no standard definition for the term "components". A number of proposals exist in the literature, *e.g.* [26, 152, 159, 192, 198], with emphasis on different aspects of components. In this thesis, we adopt Xiong's proposal [198], looking at components at an abstract level. That is, a component encapsulates state and behaviour, and interacts with its environment through its interfaces. The interfaces are the entrance points to access the services provided by the component [192]. In particular, we consider a component as a reusable and independent unit of specification, subject to composition.

Together with remarkable benefits gained by advocating independent development of components, component-based design also brings us new problems. These problems include "how can we be sure that these independently developed components work together?", and further "how can we know that a component-based system does what we want it to do?". Solving these problems is essential to ensure the functionality and quality of the resultant systems. Basically, this involves issues at four levels.

1. **Data type compatibility:** meaning that communicating components agree on the types of data being exchanged. For example, the sender should not send strings if only integers are expected by the receiver.

2. **Behaviour type compatibility:** meaning that interacting components described in different models of computation agree on common communication schemas [198]. According to Lee [134], a model of computation describes the "laws of physics" that govern component interactions. Components written in different formalisms may have different models of computation. For example, components in the CSP domain rely on rendezvous synchronisation, while Kahn process networks are stream-based. The compatibility at this level ensures the compatibility between the interaction laws of components. In contrast to behaviour compatibility (below), this level of compatibility usually can be statically determined, since the models of computation of components are usually predefined.

3. **Behaviour compatibility:** meaning that there exists no unexpected interaction between components. This ensures that the context dependency of every component is fulfilled. More specifically, since a component and its client are usually developed in isolation, there often exists a contract to ensure safe interactions between them. The contract depicts the communication protocol that a component assumes about its context. In order for the component to function properly, the environment has to abide by this protocol.

4. **Proving desired properties:** meaning that the system does what we wanted it to do. This involves checking safety and liveness requirements of the system. For instance, a typical requirement may be deadlock freedom. Deadlock refers to a situation where every component is waiting for an event that will never happen.

In the existing industrial standards, such as OMG CORBA [158], Microsoft COM [153] and Sun JavaBeans [190], which are basically distributed object-oriented models, component interfaces only capture syntactic information but leave out important semantic information. For example, the specification of dynamic properties and semantic constraints of components at their interfaces are not directly supported [198]. Thus these standards themselves do not provide facilities to answer questions at the last three levels.

A body of research on these issues has been explored in the literature. Xiong in his PhD thesis [198] has recognized and intensively studied issues at the first two levels, *i.e.* data type and behaviour type compatibility. Issues at the third level, *i.e.* behaviour compatibility, have been studied in [46, 72, 132, 168, 174, 201]. Further, research based on the existing industrial standards attempts to support the higher level analysis of component-based systems by enhancing the component interfaces with semantic information. Examples include [187, 15, 126, 29, 172, 34].

In this thesis, we concentrate on the last two levels of issues, *i.e.* behaviour compatibility and desired properties. To do so, we make some simplifications at the first two levels. We enhance the formalisms, used for describing components, with typed input/output ports (or channels). In particular, ports are embedded into formalisms to represent the service access points, capturing necessary syntactic information of components. We then restrict the communication between components to asynchronous communications via ports, as distributed systems are the focus of attention. In this way, we simplify the behaviour type compatibility problem to the adaptability of formalisms for supporting this common communication schema, and thus circumvent the need for explicitly checking the behaviour type compatibility between components. Although this makes a compromise on the flexibility of the composition mechanism, we pave the way towards answering questions at higher levels.

Furthermore, like [198], we have chosen interface automata [54] for describing compatibility requirements of components, due to the lightweight and formal nature of the language. Different from [198] where interface automata are used to ensure the behaviour type compatibility between components, we use the language to capture a higher level of compatibility, namely, behavioural compatibility.

2.4 Modelling Languages

In the development of a concurrent system, it is especially important for the designer to have a precise description of the system behaviour. Traditional description methods based on informal or semi-formal languages often are not good enough to achieve this. To cope with this difficulty, a large number of formal notations and techniques have been proposed, e.g. Z [186], LOTOS [23] and process algebras such as CSP [94] and CCS [154]. They provide systematic approaches to the construction of unambiguous specification models and the automated verification of their desired properties. The use of formal languages greatly enhances clarity of the design process and confidence in the correctness of the resulting systems.

However, these benefits are usually counterbalanced by the fact that application and software engineers commonly lack experience in interpreting and manipulating the complex underlying mathematical formalisms. Furthermore, there is a clear need in practice for development techniques being intuitive and suggestive, so that the designer can focus on the specified problem rather than coping with the specification formalism. This is also important for successful communication between the designer and the customer, especially in the requirements elicitation stage.

Visual languages appear to be a promising approach to overcome these drawbacks. Outstanding examples include dataflow diagrams [183, 135], Petri nets [176], Statecharts [87], SDL [106], MSC [107], and UML [159]. In these languages, the intricate mathematical fundamentals are hidden and transformed into bubbles, arcs, and their graphical (or physical) relations. Consequently, they not only allow the formal specification and verification of systems, but support intuitive understanding of systems owing to multiple dimensions of the graphs. In recent years, visual languages have gained a wide acceptance in industry for designing, documenting, and even programming software. There also exist notable examples where visual languages are used in the structured analysis and graphical representation of textual formalisms [20].

Therefore, towards the goal of being lightweight and practical, in this thesis we focus on a class of visual languages, viz. *graph-like* notations, for the specification of discrete-event systems. These notations consist of vertices (represented as closed geometrical shapes) and edges (represented as lines) connecting vertices. Examples include Petri nets, finite state diagrams, statecharts (both Harel and UML variants), etc. Among them, Petri nets and UML statecharts will be further investigated in Chapter 4.

3 Compositional Verification of Component-Based Systems

In recent years, component-based development has become more popular for the production of large-scale computer applications. By building systems from independently developed components, a promising means of achieving software reuse, rapid development and complexity management is provided. However, due to the state space explosion, the applicability of exhaustive analysis is largely limited. Among various techniques attacking this problem, compositional verification [10, 79, 92, 93, 150] is a powerful divide-and-conquer technique best matching the modular nature of component-based systems. It decomposes the verification task of a system into sub-tasks of individual components and addresses these sub-tasks independently. The key to this is to consider each component in conjunction with the assumptions about its context, and to consider the composition of components in conjunction with their interface behaviour.

In component-based systems, however, this key information is often missing or only informally described. Currently, the interface specifications of components tend to be rather restricted, capturing only the *signatures*, i.e. the names, data types and direction of information flow, but excluding information about the *communication protocols* of components. This is because software engineers lack a formal means for precisely specifying the interfaces behind which components encapsulate their services. As a result, components cannot be verified independently due to the lack of information about the environments in which they are embedded, and the composition of components cannot be verified due to the lack of rigorous specifications of the interface behaviour of components.

In this chapter, we present a formal technique of compositional verification which focuses on communication protocols while abstracting away from the data values being communicated. The protocol of a component describes the services it provides, the way it reacts to its inputs and what it expects from its environment. It does not, however, disclose the implementation detail of the component. We use interface automata (IAs), a formal lightweight language proposed by de Alfaro and Henzinger [54], as the notation for describing protocols.

With the contextual assumptions captured by an interface automaton (IA), each component can be checked for conformance with the IA in isolation from the system. This ensures that a component is able to abide by the interaction protocol given by the IA, provided its environment behaves as expected. Furthermore, the composition of components can be verified utilising the interface behaviour of components specified by the IAs while disregarding the internal activities of components. Using this divide-and-conquer approach, the state space explosion problem can be alleviated.

The remainder of this chapter is structured as follows. In Section 3.1, the underlying concepts such as general reactive systems and networks are introduced. In Section 3.2, discrete-event components and interface automata are defined followed by the conformance relations between them. In Section 3.3, networks of discrete-event components are clarified and compositional verification methods for determining their consistency, deadlock freedom and safety properties are presented. Finally, Section 3.4 presents a summary and Section 3.5 compares our approach with the related work.

3.1 Preliminaries

This section introduces general definitions for reactive transition systems and networks composed of reactive transition systems. These definitions are set in the framework of Arnold and Nivat. Subsequent sections will specialise these definitions for discrete event components and interface automata.

3.1.1 Reactive Transition Systems

Definition 1. A *reactive transition system* (RTS) is defined as $L = (s^0, S, \Sigma, \Delta)$, where

- S is a set of states and $s^0 \in S$ is the initial state;
- Σ is a set of events, consisting of three mutually disjoint sets of input events Σ^I, output events Σ^O, and internal events Σ^H;
- $\Delta \subseteq S \times \Sigma \times S$ is a set of steps.

RTSs are similar to labelled transition systems (LTSs) [13] which have been used to give the operational semantics of many modelling languages, e.g. CSP [177] and CCS [154]. RTSs differ from LTSs in having an explicit distinction between input, output and internal events (called labels in LTSs). The distinction reflects the fact that, in an asynchronous distributed application, a system has control over its internal and output events, but no control over its input events. Instead, when an input event occurs is under the control of the environment. That is, a system decides when to produce an output, while the environment decides when to provide an input. Hence we let $\Sigma^{ctrl} = \Sigma^O \cup \Sigma^H$ be a set of controllable events of L and $\Sigma^{obs} = \Sigma^I \cup \Sigma^O$ be a set of observable events.

A RTS L is *finite* if both S and Σ are finite. L is *deterministic* if $\forall e \in \Sigma, s, s', s'' \in S$, $(s, e, s'), (s, e, s'') \in \Delta$ implies $s' = s''$. Otherwise L is *nondeterministic*. Further, a state $s \in S$ is said to be a *terminal state* if $\nexists (s, e, s') \in \Delta$.

Definition 2. A *trace* σ of a RTS L from $s_1 \in S$ is an event sequence $e_1 e_2 \dots e_m$ such that $\exists s_1, \dots, s_{m+1} \in S, \forall j\colon 1 \le j \le m, (s_j, e_j, s_{j+1}) \in \Delta$. σ is said to be *internal* if $\forall j\colon 1 \le j \le m$, $e_j \in \Sigma^H$, or to be *empty* if $m = 0$. An empty trace is denoted as λ. Given a set of events E, the event restriction $\sigma \upharpoonright_E$ is an event sequence obtained by removing from σ all events not in E.

Definition 3. Let σ, m, $e_1, \dots, e_m$ and $s_1, \dots, s_{m+1}$ be as in Definition 2. Then s_{m+1} is called *reachable from* s_1 *(via* σ*)*. Further, s_{m+1} is called *reachable in* L if $s_1 = s^0$, or *reachable from* s_1 *by* a set of events E if $e_j \in E$ for all $j\colon 1 \le j \le m$.

In the following, we write $s \xrightarrow{e} s'$ as a shorthand for $(s, e, s') \in \Delta$, and $s \overset{e}{\Longrightarrow} s'$ for $e \in \Sigma^{obs}$ if $\exists s'' \in S$, $s'' \xrightarrow{e} s'$ and s'' is reachable via a (possibly empty) internal trace of L from s.

Definition 4. Given a state $s \in S$, the sets of *enabled input* and *output events* at s are defined by $\mathfrak{en}^I(s) = \{e \in \Sigma^I \mid \exists s' \in S, s \xrightarrow{e} s'\}$ and $\mathfrak{en}^O(s) = \{e \in \Sigma^O \mid \exists s' \in S, s \overset{e}{\Longrightarrow} s'\}$, respectively. An input event $e \in \Sigma^I$ is called *refused at* s if $e \notin \mathfrak{en}^I(s)$. A RTS L is called *input-universal* if $\forall s \in S$, $\mathfrak{en}^I(s) = \Sigma^I$.

In our approach, it is important to develop two derivatives for RTSs: mirrors and input-universal versions. Mirrors will provide the ideal environments for RTSs, while input-universal versions will make RTSs able to accept all input events with the addition of trap and idle steps.

Definition 5. Given a RTS L, the *mirror* of L is a RTS $M = (s_L^0, S_L, \Sigma_M, \Delta_L)$, where $\Sigma_M^I = \Sigma_L^O$ and $\Sigma_M^O = \Sigma_L^I$.

The mirror of L is identical to L but has the input and output events of L interchanged.

Definition 6. Consider a RTS L. Let

- $\bot \notin S_L$ be a single error state,
- $\Delta_{trap} = \{(s, e, \bot) \mid s \in S_L, e \in \Sigma_L^I \setminus \mathfrak{en}_L^I(s)\}$ be a set of trap steps, and
- $\Delta_{idle} = \{(\bot, e, \bot) \mid e \in \Sigma_L^I\}$ be a set of idle steps.

Then the *input-universal version* of L is a RTS U such that

$$U = \begin{cases} (s_L^0, S_L \cup \{\bot\}, \Sigma_L, \Delta_L \cup \Delta_{trap} \cup \Delta_{idle}) & \text{if } \Delta_{trap} \neq \emptyset, \\ L & \text{otherwise.} \end{cases}$$

The input-universal version is a RTS that includes not only the existing steps but also new trap steps. A trap step, taken when an unspecified input event is received, will cause the error state $\bot$ to be entered. Clearly, for every state-event pair (s, e) with $s \in S$ and e a refused input event at s, a step is added to Δ_{trap} which emanates from s, receives e and enters $\perp$. In addition, we have ensured that the input-universal version of an (already) input-universal RTS is the same RTS.

3.1.2 RTS Networks

We define the composition of RTSs in terms of synchronisation vectors introduced by Arnold and Nivat [13]. Their approach to synchronisation was quite general in that any non-empty set of processes may synchronise on the events named in so-called *synchronisation vectors*. In this way, not only peer-to-peer but also multicast and broadcast communication among processes can be described. Our approach differs from Arnold and Nivat in differentiating input and output events, since this is an important issue for distributed component-based systems. Components should be able to accept inputs at any time, but may be selective in when they produce outputs. Also, we insist that each synchronisation vector should have exactly one output event and an arbitrary number of synchronised input events. To define RTS networks, we introduce a preliminary definition to help clarify the nature of synchronisation vectors.

Definition 7. Consider sets $E_0, E_1, \dots E_n$, a relation $R \subseteq \Pi_{0 \le i \le n} E_i$, and a set $E \subseteq \bigcup_{0 \le i \le n} E_i$ such that $E \cap E_i \cap E_j = \emptyset$ for all $i, j \colon 0 \le i, j \le n \wedge i \neq j$. Let projections $\pi_i \colon R \longrightarrow E_i$ for $0 \le i \le n$ and sets of keys $\kappa_{\mathrm{r}} = \{\pi_i(\mathrm{r}) \in E \mid 0 \le i \le n\}$ for $\mathrm{r} \in R$. Then R is said to be *indexed by E* if there exists:

- exactly one key per tuple: $|\kappa_{\mathrm{r}}| = 1$ for all $\mathrm{r} \in R$;
- at most one occurrence per key: $\forall e \in E, \exists \mathrm{r} \in R, e \in \kappa_{\mathrm{r}}$ implies $\forall \mathrm{r}' \in R \setminus \{\mathrm{r}\}, e \notin \kappa_{\mathrm{r}'}$.

We will use such a relation to capture the synchronisation patterns between RTSs. These patterns may have multiple input events but exactly one output event, which will then individually provide a key and cumulatively provide the index for the relation. Also, an output event is involved in at most one synchronisation pattern. In defining RTS networks, we assume a special symbol ϵ which is not an event of any RTS.

Definition 8. A *RTS network* is a tuple $N = (\Sigma, W, R)$, where

- Σ is a set of *external* events of the network, consisting of two disjoint sets of input events Σ^I and output events Σ^O. We let $\Sigma^\sharp = \Sigma \cup \{\epsilon\}$;
- W is a finite set of RTSs. We let $\Sigma_l^\sharp = \Sigma_l^{obs} \cup \{\epsilon\}$ for all $l \in W$;
- $R \subseteq \Sigma^\sharp \times \Pi_{l \in W} \Sigma_l^\sharp$ is a relation indexed by $\Sigma^I \cup \bigcup_{l \in W} \Sigma_l^O$.

A RTS network is called *closed* if $\Sigma = \emptyset$, or *open* otherwise. We often write $N = (W, R)$ for a closed RTS network N. The relation R is called a set of *synchronisation vectors* (in the terminology of the Arnold and Nivat model). A synchronisation vector in the set describes a particular synchronisation pattern between the component RTSs. More specifically, the RTSs with events present in the vector are synchronised, while the other RTSs marked by ϵ remain unchanged. The symbol ϵ represents the irrelevancy of a RTS to a particular synchronisation. Further, the RTS corresponding to the network events is referred to as the environment, denoted by "env". Its output events denote the network input events and its input events denote the network output events, *i.e.* $\Sigma_{\mathsf{env}}^O = \Sigma^I$ and $\Sigma_{\mathsf{env}}^I = \Sigma^O$. Note that env is an unknown RTS and thus we only use it as a syntactic term to facilitate further explanation.

In order to give RTS networks a sound interleaving semantics, we require exactly one output event in each synchronisation vector, because output is nonblocking in asynchronous applications. Also, we require that at most one synchronisation vector can be matched for a particular output event. Hence the synchronisation vectors R are indexed by the output events (of the environment or the component RTSs). In each synchronisation vector, the unique output event is called the *produced event* and the input events in the vector are the *consumed events* of the synchronisation. Also, the RTS that produces the produced event is called the *producer*, the RTSs that receive the consumed events are the *consumers*, and the rest are the *idlers* of the synchronisation. A formal specification is given as follows.

Definition 9. Consider a RTS network N and a synchronisation vector $\mathrm{r} \in R$. Let $W' = \{\mathsf{env}\} \cup W$, $l \in W'$, and projections $\pi_l \colon R \longrightarrow \Sigma_l$ for all l. Then the *producer* of r, denoted by ρ_{r}, is the unique RTS l such that $\pi_l(\mathrm{r}) \in \Sigma_l^O$. $\pi_{\rho_{\mathrm{r}}}(\mathrm{r})$ is called the *produced event* of r. Also, the set of *consumers* of r, denoted by η_{r}, consists of all RTSs l such that

$\pi_l(\mathrm{r}) \in \Sigma_l^I$. For any consumer l, the event $\pi_l(\mathrm{r})$ is called a *consumed event* of r. In addition, an *idler* of r is a RTS l such that $\pi_l(\mathrm{r}) = \epsilon$. The set of idlers is denoted by ι_{r}.

Before giving the semantics of general RTS networks, we first consider networks of input-universal RTSs.

Definition 10. Consider a RTS network $N = (\Sigma_N, W, R)$ such that all $l \in W$ are input-universal. The *synchronised product* of N is a RTS $L = (s^0, S, \Sigma_L, \Delta)$, where

- $s^0 = \Pi_{l\in W} s_l^0$ and $S \subseteq \Pi_{l\in W} S_l$ is the smallest set such that $s^0 \in S$ and $\forall \mathrm{s} \in S, (\mathrm{s}, e, \mathrm{s}') \in \Delta$ implies $\mathrm{s}' \in S$. We assume projections $\pi_l \colon S \longrightarrow S_l$ and let $\mathrm{s}_l = \pi_l(\mathrm{s})$ and $\mathrm{s}'_l = \pi_l(\mathrm{s}')$ for $l \in W$, $\mathrm{s}, \mathrm{s}' \in S$;
- $\Sigma_L^I = \Sigma_N^I$, $\Sigma_L^O = \Sigma_N^O$ and $\Sigma_L^H \subseteq \bigcup_{l\in W} \Sigma_l^{ctrl}$;
- Δ consists of input steps

$$\begin{aligned} &\{(\mathrm{s}, e, \mathrm{s}') \mid e \in \Sigma_L^I, \exists \mathrm{r} \in R, \rho_{\mathrm{r}} = \mathsf{env} \wedge e = \pi_{\mathsf{env}}(\mathrm{r}) \wedge \delta(\mathrm{s}, \mathrm{r}, \mathrm{s}')\}, &(3.1)\\ \cup\ &\{(\mathrm{s}, e, \mathrm{s}) \mid e \in \Sigma_L^I, \nexists \mathrm{r} \in R, \rho_{\mathrm{r}} = \mathsf{env} \wedge e = \pi_{\mathsf{env}}(\mathrm{r})\}, &(3.2)\end{aligned}$$

output steps

$$\begin{aligned} \{(\mathrm{s}, e, \mathrm{s}') \mid e \in \Sigma_L^O, \exists l \in W, \mathrm{r} \in R, l = \rho_{\mathrm{r}} \wedge e = \pi_{\mathsf{env}}(\mathrm{r})\\ \wedge (\mathrm{s}_l, \pi_l(\mathrm{r}), \mathrm{s}'_l) \in \Delta_l \wedge \delta(\mathrm{s}, \mathrm{r}, \mathrm{s}')\}, \end{aligned} \tag{3.3}$$

and internal steps

$$\begin{aligned} &\{(\mathrm{s}, e, \mathrm{s}') \mid \exists l \in W, e \in \Sigma_l^H \wedge (\mathrm{s}_l, e, \mathrm{s}'_l) \in \Delta_l \wedge (\forall g \in W \setminus \{l\}, \mathrm{s}'_g = \mathrm{s}_g)\} &(3.4)\\ \cup\ &\{(\mathrm{s}, e, \mathrm{s}') \mid \exists l \in W, e \in \Sigma_l^O \wedge \exists \mathrm{r} \in R, e = \pi_l(\mathrm{r}) \wedge \pi_{\mathsf{env}}(\mathrm{r}) = \epsilon\\ &\qquad \wedge (\mathrm{s}_l, e, \mathrm{s}'_l) \in \Delta_l \wedge \delta(\mathrm{s}, \mathrm{r}, \mathrm{s}')\} &(3.5)\\ \cup\ &\{(\mathrm{s}, e, \mathrm{s}') \mid \exists l \in W, e \in \Sigma_l^O \wedge (\nexists \mathrm{r} \in R, e = \pi_l(\mathrm{r}))\\ &\qquad \wedge (\mathrm{s}_l, e, \mathrm{s}'_l) \in \Delta_l \wedge (\forall g \in W \setminus \{l\}, \mathrm{s}'_g = \mathrm{s}_g)\}, &(3.6)\end{aligned}$$

where $\delta(\mathrm{s}, \mathrm{r}, \mathrm{s}') = (\forall g \in \eta_{\mathrm{r}}, (\mathrm{s}_g, \pi_g(\mathrm{r}), \mathrm{s}'_g) \in \Delta_g) \wedge (\forall h \in \iota_{\mathrm{r}}, \mathrm{s}'_h = \mathrm{s}_h)$.

According to the definition, such a RTS network defines a RTS (later called the *composite RTS*). A state of the RTS is a vector of states of all its component RTSs (later called the *components*), and its initial state is a vector of initial states of the components. An input step of the composite RTS is a step receiving a network input event. If there exists a synchronisation vector where env is the producer and the event is the one produced, the input step also synchronises the corresponding input steps of the consumers (formula 3.1). Otherwise, the composite RTS remains in the same state after consuming the input event (formula 3.2). Further, the composite RTS will perform an output step when the producer of a synchronous vector generates the produced event, provided that env is a consumer of the vector (formula 3.3). The step involves the corresponding output step of the producer synchronised with the corresponding input steps of the consumers including the environment. Finally, an internal step taken by any component is an internal step of the composite RTS (formula 3.4). This step clearly has no impact on other components or the environment. Also, an output step taken by a component becomes internal to the composite RTS, when the corresponding synchronisation vector involves no network events (formula 3.5) or no corresponding vector exists (formula 3.6). Similarly, the step described in formula 3.5 will synchronise the corresponding input steps of the consumers.

One can see that the composite RTS involves not only synchronised communications among the components but also interleavings of their internal activities. The latter are the main contributors to the state space explosion when a network is directly analysed and thus our approach seeks to minimise their effect.

We now generalise the synchronised product to an arbitrary RTS network.

Definition 11. Consider a RTS network $N = (\Sigma, W, R)$. Let $N' = (\Sigma, W', R)$ be a derived RTS network such that $W' = \{U_l \mid l \in W\}$, where U_l is the input-universal version of l. Then the *synchronised product* of N is defined by the synchronised product of N' (Definition 10).

The above definition acknowledges the fact that in asynchronous systems the reception of an unspecified input event often indicates a design error.

Immediately from Definition 10 and 11, we can get the following proposition.

Proposition 1. *The synchronised product of any RTS network is input-universal.*

To relate the behaviour of a RTS network to that of its components, we define trace projections on components.

Definition 12. Consider a RTS network $N = (\Sigma, W, R)$ and a RTS $l \in W$. Let L_N be the synchronised product of N, σ a trace of L_N from s_N^0, and

$$\psi = \{(o, i) \mid \exists \mathrm{r} \in R, l' \in \{\mathsf{env}\} \cup W, o \in \Sigma_{l'}^O, \pi_{l'}(\mathrm{r}) = o \land i \in \Sigma_l^I, \pi_l(\mathrm{r}) = i\}$$

a function mapping an output event of any other RTS, which can cause the RTS l to take an input step, into the corresponding input event of l. Then the *trace projection* $\pi_l(\sigma)$ *of* σ *on* l is defined by

$$\pi_l(\sigma) = \begin{cases} \lambda & \text{if } \sigma = \lambda, \\ e \cdot \pi_l(\sigma') & \text{if } \sigma = e \cdot \sigma' \land e \in \Sigma_l^{ctrl}, \\ i \cdot \pi_l(\sigma') & \text{if } \sigma = e \cdot \sigma' \land i \in \Sigma_l^I \land (e, i) \in \psi, \\ \pi_l(\sigma') & \text{if } \sigma = e \cdot \sigma' \land e \notin \Sigma_l^{ctrl} \land \nexists (e, i) \in \psi. \end{cases}$$

The trace projection $\pi_l(\sigma)$ on l is an event sequence consisting of the events that l takes while the network N follows trace σ from its initial state s_N^0. It is computed recursively by considering each event e in σ in turn. If e is an internal or output event of l, then e is appended to the trace projection; if e is an output event of other RTSs which corresponds to a synchronisation vector r with $\pi_l(\mathrm{r})$ an input event of l, then $\pi_l(\mathrm{r})$ is appended to the trace projection; otherwise e is ignored. Note we have ensured that $\pi_l(\sigma)$ is a trace of l from s_l^0.

3.2 Independent Analysis of Components

In this section, general RTSs are specialised as discrete-event components and as interface automata. Also, the conformance relations between these are studied and then a practical conformance checking method is presented.

3.2.1 Discrete-Event Components

As RTSs are often too elementary for practical use, we extend these systems in two ways. First of all, we redefine an event to consist of two parts including a kind and a value, and associate each part with different importance depending on the context. Kinds are used to classify events while values represent data being communicated (also called *event parameters*). This allows us to introduce a level of abstraction to component behaviour and vary the level of abstraction by changing the way events are partitioned. In addition, we extend RTSs with input/output ports, each port identifying a particular kind of events. Then it can be assumed that a component (or a reactive system) communicates with others only through its ports. That is, a component always receives data (or messages) fed to its input ports and produces data (or messages) via its output ports. In this way, event kinds (or ports) play a critical role in defining component composition but have little impact on the computation of individual components, while event values are of great importance to component computation but less relevant to component composition. This facilitates the system modelling in that the ports can form a component's view of the rest of the system and decouple the outside world from the component. It thus largely reduces the interdependency between components. This approach also allows the designers to compose components by simply connecting ports, and thus relieves them from the labor of relating individual events. In defining components, we assume a countable universe of transmitted values $\mathcal{V}$.

Definition 13. A *discrete-event component* (DEC) is defined as $C = (\alpha, \theta, s^0, S, \Sigma, \Delta)$, where

- α is a finite set of ports, consisting of two disjoint sets of input ports α^I and output ports α^O;
- $\theta \colon \alpha \longrightarrow 2^{\mathcal{V}}$ is a total function, mapping each port to a subset of values from the universe;
- $\Sigma = \Sigma^I \cup \Sigma^O \cup \Sigma^H$ is a set of events, where $\Sigma^I = \{\langle f, v\rangle \mid f \in \alpha^I, v \in \theta(f)\}$, $\Sigma^O \subseteq \{\langle f, v\rangle \mid f \in \alpha^O, v \in \theta(f)\}$ and $\Sigma^H \cap (\alpha \times \mathcal{V}) = \emptyset$;
- The tuple (s^0, S, Σ, Δ) forms an input-universal RTS.

A DEC is called *closed* if $\alpha = \emptyset$, or *open* otherwise. θ is called the *typing* function, associating each port with a type (or kind) of values that can be transmitted via the port. An input/output event of a DEC is regarded as an occurrence of message transfer at an input/output port of the DEC, while internal events of each DEC are considered to be unique to that DEC. Also, like [144, Chapter 8], we require DECs to be input-universal RTSs. This acknowledges the fact that components are often developed to work properly in unknown environments in bottom-up design. This is also a requirement for independent deployment of components. In the following, we abbreviate $\langle f, v\rangle$ to $f.v$ for $\langle f, v\rangle \in \Sigma^I \cup \Sigma^O$.

Figure 3.1 models a DEC in a compact form, where black bullets, arcs and triangles represent states, steps and ports, respectively. In particular, the initial state is pointed to by an arrow with no source. Input ports are listed at the left, *e.g.* "a" and "b", and output ports at the right, *e.g.* "c". Following the port names, types of transmitted values are specified. In this case, all values must be integers. In addition, the text attached to an arc denotes a parameterized event. For instance, $a.x?$ denotes an event receiving from a an integer represented by x, while $c.(x+y)!$ indicates an event producing via c an integer, which is the sum of x and y. Note that the actual size of the DEC's state space depends on the ranges of x and y.

This example DEC describes an adder which accepts two parameters respectively from the two input ports and then reports their sum via the output port. The adder expects the environment to behave in a certain way in order to function properly. For instance, it expects to be fed with only one parameter of each kind before producing the

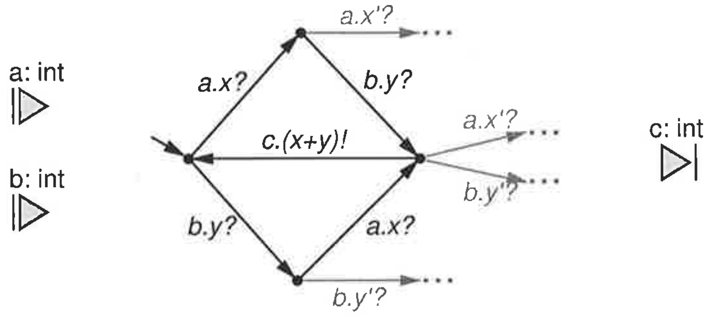


Figure 3.1: An adder DEC model

sum. However, it is often required for a component to be robust and able to work in all kinds of environments. Hence this example component also performs some tasks such as error reporting or recovery (denoted by grey arcs) if the environment does not behave as it expects, *e.g.* providing two values of kind a or b in a row. Since the way these tasks are specified is not the focus of this thesis, we have omitted the detail.

It should be noted that even though robustness is required of DECs, the system designer often does not want the "grey" tasks to be executed, since it prevents the DECs from providing their normal functionality to the system. Also, in practice the portion of a DEC providing the normal functionality is usually intensively tested or verified, whereas the other portion is not and thus could contain potential bugs. Once encountered, these bugs may cause problems, sometimes even disastrous consequences. These concerns highlight one of the main goals of this thesis, which is to ensure that the branches irrelevant to the functionality needed in a component-based design, *e.g.* the grey tasks, will never be executed.

3.2.2 Interface Automata

Assembling events by kinds makes it possible to specify the interaction protocols expected by the designer of components in terms of temporal relations between different kinds of events, while abstracting away from the specific values. The protocols are very useful for guiding the development or selection of individual components and for further study of a system design, and thus should be formally specified. We employ a restricted version of *interface automata* [54] to serve this purpose. We constrain interface automata to be deterministic and to have no internal events. We believe that this version is sufficiently expressive for our purpose and, as shown later, this restriction leads to an efficient conformance checking method.

Definition 14. An *interface automaton* (IA) is defined as a finite deterministic RTS $A = (s^0, S, \Sigma, \Delta)$, where $\perp \notin S$ and $\Sigma^H = \emptyset$. We let $\mathcal{U}^{ia}$ be a universal set of IAs.

Let the events of IAs correspond to the ports of DECs. Then the information conveyed by an IA is twofold. On the one hand, it restricts the kinds of output events that a component under consideration can produce. On the other hand, it states the component's

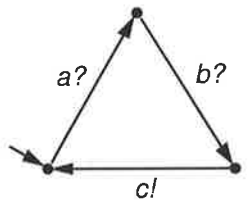


Figure 3.2: An adder IA

assumption that the environment never provides input events of an unspecified kind. As an example, suppose we get an IA shown in Figure 3.2 when decomposing a system. Then an assumption is captured that the environment always provides an event of kind a followed by an event of kind b and then waits for an event of kind c before providing any further events. Also, it is guaranteed that the component under development or selection must not produce any output before getting both a and b.

Compared with Figure 3.1, we can learn that the DEC can be an implementation of this IA in that the DEC does not break the guarantee expressed by the IA and can be used in an application where the environment does not break the assumption captured by the IA. We shall formally study this relationship in a general context in the next section. Moreover, we emphasize that IAs are intermediate products of top-down design capturing the interface and the protocol but not the implementation, while DECs are building blocks for bottom-up design.

While abstracting away implementation details such as the data being communicated, IAs can describe the interaction protocols expected of components at a high level of abstraction. We shall show that the use of IAs enables the compositional verification of component-based systems. It is worth noting that the abstraction is a relative issue in that the level of abstraction heavily depends on the way events are partitioned. The level of abstraction becomes greater as we classify more events into each kind.

To facilitate further study, we define for IAs the *most abstract implementations*, which include the IA events as ports and are able to produce all possible data values accompanying an enabled IA output event.

Definition 15. Consider an IA A and a total typing function $\theta\colon \Sigma_A \longrightarrow 2^{\mathcal{V}}$. Let U be the input-universal version of A, then the *most abstract implementation (MAI)* of A (with respect to θ) is a DEC $J = (\alpha_J, \theta, s_U^0, S_U, \Sigma_J, \Delta_J)$, where

- $\alpha_J^I = \Sigma_A^I$ and $\alpha_J^O = \Sigma_A^O$;
- $\Sigma_J^I = \{\langle f, v\rangle \mid f \in \Sigma_A^I, v \in \theta(f)\}$, $\Sigma_J^O = \{\langle f, v\rangle \mid f \in \Sigma_A^O, v \in \theta(f)\}$ and $\Sigma_J^H = \emptyset$;
- $\Delta_J = \{(s, f.v, s') \mid (s, f, s') \in \Delta_U, v \in \theta(f)\}$.

A MAI of an IA is an input-universal DEC. Similar to the input-universal version of the IA, after getting an unspecified input event with an arbitrary value, the MAI goes to the error state, where it can accept all input events but never produce outputs or move out of the error state.

3.2.3 Conformance of Discrete-Event Components

The employment of IAs for guiding component development or selection leads to an important aspect of this approach, viz. the conformance of DECs with IAs. The intention is that an IA can safely be implemented by a conforming DEC without compromising the system safety properties that hold for the IA, particularly in a network of IAs.

The conformance cannot be defined by traditional refinement relations, *e.g.* trace containment and simulation, because they only allow the implementation to have less input and output behaviour than the specification, whereas DECs are able to handle more inputs than IAs. Instead we adopt alternating simulation [54], a relation based on an optimistic view of the environment. More specifically, the implementation is always assumed to run in an environment where the assumption of the specification is respected. In this way, the implementation can offer less outputs (since it will be less likely to violate the system safety) and accept more inputs (since the environment will not offer them). In the following, we extend the relation to accommodate the implementation with data values, as required by our definition of DECs.

Definition 16. Consider an IA A and a DEC C such that $\Sigma_A^I \subseteq \alpha_C^I$ and $\Sigma_A^O \supseteq \alpha_C^O$. C *conforms to* A, written $C \prec A$, if there exists an alternating simulation relation $\prec \subseteq S_C \times S_A$ such that $s_C^0 \prec s_A^0$ and for $q \prec s$, the following conditions hold:

1. $\forall e \in \Sigma^H_C, \exists (q, e, q') \in \Delta_C$ implies $q' \prec s$;
2. $\forall f.v \in \Sigma^O_C, \exists (q, f.v, q') \in \Delta_C$ implies that $\exists (s, f, s') \in \Delta_A$ such that $q' \prec s'$;
3. $\forall f \in \Sigma^I_A, \exists (s, f, s') \in \Delta_A$ implies that $\forall v \in \theta_C(f), (q, f.v, q') \in \Delta_C$ such that $q' \prec s'$.

For a DEC state q to simulate an IA state s, first of all, the resultant DEC state must simulate the previous IA state s after the DEC takes an internal step from q (Condition 1). Also, the DEC must not produce an output event that the IA cannot produce (Condition 2). Note that Condition 1 and 2 implies that $\forall f.v \in \mathfrak{en}^O_C(q)$ such that $f \in \mathfrak{en}^O_A(s)$. Further, the resultant DEC state must simulate the resultant IA state after the DEC takes an event $f.v$ (for any value v) from q and the IA takes the event f (Condition 2 and 3). Note also that the input-universal nature of DECs implies that $\forall f \in \mathfrak{en}^I_A(s)$, $v \in \theta_C(f)$ such that $f.v \in \mathfrak{en}^I_C(q)$. Therefore, this relation encodes an input and output duality that the DEC at state q allows more input events but produces less output events than the IA at state s. Clearly, the most abstract implementation of an IA (with respect to arbitrary θ) conforms to the IA.

To support the verification of system properties beyond safety such as deadlock freedom, we develop a stronger conformance relation with an additional restriction.

Definition 17. Consider an IA A and a DEC C such that $C \prec A$. C *live-conforms to* A, written $C \preceq A$, if for all $q \in S_C, s \in S_A, q \prec s \wedge \mathfrak{en}^O_A(s) \neq \emptyset$ implies $\mathfrak{en}^O_C(q) \neq \emptyset$;

The imposed restriction requires that at state q or after some internal steps from q, C must be able to generate at least one of the output events that the IA can produce at state s. This requires a component to fulfil the output obligation specified by the IA in order to be somewhat helpful to others.

It should be noted that Definition 16 and 17 only allow a DEC with equal or less output ports to be an implementation of an IA. In practice, however, DECs often have not only more input ports but also more output ports, especially when third-party components are deployed. They usually provide more services than needed in an application domain. To solve this, we define instantiated components for these DECs and redefine the conformance relations with relaxed conditions.

Definition 18. An *instantiated component* of a DEC C with respect to a set of ports $\mathcal{O}$ is defined by $\hat{C}(\mathcal{O}) = (\alpha_{\hat{C}}, \theta_C, s_C^0, S_C, \Sigma_{\hat{C}}, \Delta_C)$, where

- $\alpha_{\hat{C}}^I = \alpha_C^I$ and $\alpha_{\hat{C}}^O = \alpha_C^O \cap \mathcal{O}$;
- $\Sigma_{\hat{C}}^I = \Sigma_C^I, \Sigma_{\hat{C}}^O = \{f.v \in \Sigma_C^O \mid f \in \mathcal{O}\}$ and $\Sigma_{\hat{C}}^H = \Sigma_C^{ctrl} \setminus \Sigma_{\hat{C}}^O$.

Note that $\hat{C}(\mathcal{O}) = C$ if $\alpha_C^O \subseteq \mathcal{O}$.

Definition 19. Consider an IA A and a DEC C such that $\Sigma_A^I \subseteq \alpha_C^I$. *$C$ conforms to A*, written $C \prec A$, if $\hat{C}(\Sigma_A^O)$ conforms to A as in Definition 16. *C live-conforms to A*, written $C \preceq A$, if $\hat{C}(\Sigma_A^O)$ live-conforms to A as in Definition 17.

3.2.4 Practical Conformance Checking

To check the conformance of a DEC to a given IA, instead of building the Cartesian product of their states as proposed in [54], we employ a two-step method which has significant benefits for tractability. To begin with, we calculate the local state space of the DEC utilising the context assumptions of the IA. We then determine the conformance by checking the state space for the absence of error states.

The local state space of the DEC is defined as the synchronised product of a closed RTS network of two components — one component is the DEC to be checked, while the other component is the most abstract implementation of the mirror of the IA. The second component thus represents the minimally helpful environment of the first which still provides all expected inputs to the first. Formally, the local state space is defined as follows.

Definition 20. Consider a DEC C and an IA A such that $\Sigma_A^I \subseteq \alpha_C^I$. Let $\theta = \theta_C \cup \{(f, \emptyset) \mid f \in \Sigma_A \backslash \alpha_C\}$, J be the MAI of the mirror of A with respect to θ (Definition 15), $W = \{C, J\}$, $R = \{(f.v, f.v) \mid f.v \in (\Sigma_C^O \cap \Sigma_J^I) \cup \Sigma_J^O\}$, and $N = \{W, R\}$ be a derived closed RTS network. Then the synchronised product of N is called the *local state space* of C with respect to A.

In the following, Theorem 1 and 2 present a conformance checking method for DECs with equal or less output ports than IAs, giving both the sufficient and necessary conditions for the two conformance relations.

Theorem 1. *Consider a DEC C and an IA A such that $\Sigma_A^I \subseteq \alpha_C^I$ and $\Sigma_A^O \supseteq \alpha_C^O$. Let θ and J be defined as in Definition 20 and $L_\otimes = \{s_\otimes^0, S_\otimes, \Sigma_\otimes, \Delta_\otimes\}$ be the local state space of C with respect to A. Then C conforms to A if and only if $\forall \mathrm{s} \in S_\otimes$, $\pi_J(\mathrm{s}) \neq \bot$.*

Proof of Sufficiency. Let a relation $\phi = \{(q, s) \in S_\otimes \mid q \in S_C, s \in S_A\}$, then we prove ϕ is an alternating simulation relation between C and A by induction. First, $(s_C^0, s_A^0) \in \phi$ because $s_A^0 = s_J^0$. Next, suppose $(q, s) \in \phi$, then

- For $e \in \Sigma_C^H$, if $\exists q \xrightarrow{e}_C q'$, then $\langle q', s\rangle \in S_\otimes$. Hence $(q', s) \in \phi$;
- For $f.v \in \Sigma_C^O$, we know $f \in \Sigma_A^O$, $v \in \theta(f)$ and thus $f.v \in \Sigma_J^I$. If $\exists q \xrightarrow{f.v}_C q'$, then $\exists s' \in S_J$, $\langle q, s\rangle \xrightarrow{f.v}_\otimes \langle q', s'\rangle$ (because J is input-universal). Since $\langle q', s'\rangle \in S_\otimes$, from the condition of the theorem, we have $s' \neq \bot$. Hence $s' \in S_A$ and $\langle q', s'\rangle \in \phi$;
- For $f \in \Sigma_A^I$, we know $f \in \alpha_C^I$. If $\exists s \xrightarrow{f}_A s'$, we have $\forall v \in \theta_C(f), f.v \in \Sigma_J^O \wedge s \xrightarrow{f.v}_J s'$. Since $f.v \in \Sigma_C^I$ and C is input-universal, $\exists q' \in S_C$, $\langle q, s\rangle \xrightarrow{f.v}_\otimes \langle q', s'\rangle$. Hence $\langle q', s'\rangle \in \phi$;

Therefore, ϕ is an alternating simulation relation and thus C conforms to A due to Def. 16.

Proof of Necessity. Let $\prec$ be an alternating simulation relation between C and A, σ be a trace of $L_\otimes$ from $s_\otimes^0$, and $\langle q, s\rangle \in S_\otimes$ be a state reachable via σ. Then we prove $s \neq \bot$ and $q \prec s$ by induction on the length of σ. First, when $\sigma = \lambda$, we know $\langle q, s\rangle = \langle s_C^0, s_A^0\rangle = s_\otimes^0$. Hence $s \neq \bot \wedge q \prec s$. Next, suppose $s \neq \bot \wedge q \prec s$ holds for any σ. Since $S_J = S_A \cup \{\bot\}$, we know $s \in S_A$.

- For $e \in \Sigma_C^H$, if $\exists \langle q, s\rangle \xrightarrow{e}_\otimes \langle q', s'\rangle$, then $s' = s$ (thus $s' \neq \bot$) and $q \xrightarrow{e}_C q'$. Since $q \prec s$, we can get $q' \prec s$ (Def. 16, Cond. 1).
- For $f.v \in \Sigma_C^O$, if $\exists \langle q, s\rangle \xrightarrow{f.v}_\otimes \langle q', s'\rangle$, then $q \xrightarrow{f.v}_C q'$. Since $q \prec s$ and A is deterministic, $s' \in S_A, s \xrightarrow{f}_A s'$ and $q' \prec s'$ (Def. 16, Cond. 2). Thus $s' \neq \bot$.
- For $f.v \in \Sigma_J^O$, if $\exists \langle q, s\rangle \xrightarrow{f.v}_\otimes \langle q', s'\rangle$, then $s \xrightarrow{f.v}_J s'$. Since $f \in \Sigma_A^O$, $s \xrightarrow{f}_A s'$ and $s' \neq \bot$. From Def. 16 (Cond. 3), we know $q' \prec s'$.

Therefore, that C conforms to A implies $\forall \langle q, s\rangle \in S_\otimes, s \neq \bot$. □

Theorem 2. *Consider a DEC C and an IA A such that $\Sigma_A^I \subseteq \alpha_C^I$ and $\Sigma_A^O \supseteq \alpha_C^O$. Let θ and J be defined as in Definition 20 and $L_\otimes$ be the local state space of C with respect to A. Then C live-conforms to A if and only if $\forall \mathrm{s} \in S_\otimes$, $\pi_J(\mathrm{s}) \neq \bot$ and any of the following holds:*

- *$\nexists e \in \Sigma_J^I$ such that $(\pi_J(\mathrm{s}), e, s') \in \Delta_J$;*
- *$\exists e_1, \dots, e_{m-1} \in \Sigma_C^H, e_m \in \Sigma_J^I, \mathrm{s}' \in S_\otimes$ such that s' is reachable from s via trace $e_1 e_2 \dots e_m$ in $L_\otimes$.*

Proof of Sufficiency. Because $\forall \mathrm{s} \in S_\otimes$, $\pi_J(\mathrm{s}) \neq \bot$, we know C conforms to A from Thm. 1 and $\pi_C(\mathrm{s}) \prec \pi_J(\mathrm{s})$. Suppose $\mathfrak{en}_A^O(\pi_J(\mathrm{s})) \neq \emptyset$, then we have $\exists f \in \Sigma_A^O, (\pi_J(\mathrm{s}), f, s') \in \Delta_A$. Thus $\forall v \in \theta(f)$, $(\pi_J(\mathrm{s}), f.v, s') \in \Delta_J$. Due to conditions of this theorem, we know $\exists e_1, \dots, e_{m-1} \in \Sigma_C^H, e_m \in \Sigma_J^I$, $\mathrm{s}' \in S_\otimes$ such that s' is reachable from s via trace $e_1 e_2 \dots e_m$ in $L_\otimes$. Suppose $\mathrm{s}'' \in S_\otimes$ such that s'' is reachable from s via trace $e_1 e_2 \dots e_{m-1}$ and $\mathrm{s}'' \xrightarrow{e_m}_\otimes \mathrm{s}'$. Then we have $\pi_C(\mathrm{s}'')$ is reachable from s via trace $e_1 e_2 \dots e_{m-1}$ (since $e_1, \dots, e_{m-1} \in \Sigma_C^H$) and $\pi_C(\mathrm{s}'') \xrightarrow{e_m}_C \pi_C(\mathrm{s}')$ (since input steps are blocking for DECs and thus $e_m \in \Sigma_C^O$). Therefore, $\mathfrak{en}_C^O(\pi_C(\mathrm{s})) \neq \emptyset$ and thus C live-conforms to A.

Proof of Necessity. Because C conforms to A, we have $\forall \mathrm{s} \in S_\otimes$, $\pi_J(\mathrm{s}) \neq \bot$ from Thm. 1 and $\pi_C(\mathrm{s}) \prec \pi_J(\mathrm{s})$.

- If $\nexists e \in \Sigma_J^I, (\pi_J(\mathrm{s}), e, s') \in \Delta_J$, then $\mathfrak{en}_A^O(\pi_J(\mathrm{s})) = \emptyset$.
- If $\exists f.v \in \Sigma_J^I, (\pi_J(\mathrm{s}), f.v, s') \in \Delta_J$, then $(\pi_J(\mathrm{s}), f, s') \in \Delta_A$, *i.e.* $\mathfrak{en}_A^O(\pi_J(\mathrm{s})) \neq \emptyset$. Hence we get $\mathfrak{en}_C^O(\pi_C(\mathrm{s})) \neq \emptyset$ because $C \preceq A$. We then have $\exists e_1, \dots, e_{m-1} \in \Sigma_C^H$, $e_m \in \Sigma_J^I$, $q', q'' \in S_C$ such that q'' is reachable from $\pi_C(\mathrm{s})$ via trace $e_1 e_2 \dots e_{m-1}$ in C and $q'' \xrightarrow{e_m}_C q'$. We know $\langle q'', \pi_J(\mathrm{s}) \rangle \in S_\otimes \wedge \exists s' \in S_J, \langle q', s' \rangle \in S_\otimes$ must hold (Def. 10). Therefore, we have $\langle q'', \pi_J(\mathrm{s}) \rangle$ is reachable from s via trace $e_1 e_2 \dots e_{m-1}$ in $L_\otimes$ and $\langle q'', \pi_J(\mathrm{s}) \rangle \xrightarrow{e_m}_\otimes \langle q', s' \rangle$.

In both cases, we can prove the conditions of this theorem hold. □

On the basis of the above theorems, we can now be sure that Figure 3.1 conforms to (and live-conforms to) Figure 3.2. It can thus be placed into a system design as an implementation of the latter. If we restrict ourselves to a finite set of transmitted values, the conformance can be checked by automated tools, *e.g.* our Moses analysis tools that will be implemented in Chapter 5.

The following corollary generalises these two theorems to accommodate DECs with more output ports than IAs.

Corollary 1. *Let C, A, θ and J be as in Definition 20 and $L_{\otimes}$ be the local state space of C with respect to A. Then C conforms to A if and only if $\forall \mathrm{s} \in S_{\otimes}$, $\pi_J(\mathrm{s}) \neq \bot$. In addition, C live-conforms to A if and only if $\forall \mathrm{s} \in S_{\otimes}$, $\pi_J(\mathrm{s}) \neq \bot$ and any of the following holds:*

- $\nexists e \in \Sigma_J^I$ *such that* $(\pi_J(\mathrm{s}), e, s') \in \Delta_J$;
- $\exists e_1, \ldots, e_{m-1} \in \Sigma_C^{ctrl} \setminus \Sigma_J^I$, $e_m \in \Sigma_J^I$, $\mathrm{s}' \in S_{\otimes}$, s' *is reachable from* s *via trace* $e_1 e_2 \ldots e_m$ *in* $L_{\otimes}$.

Proof. Let $\hat{C}$ represent $\hat{C}(\Sigma_A^O)$ and $\hat{L}_{\otimes}$ be the local state space of $\hat{C}$ with respect to A. Then we have $\Sigma_{\hat{C}}^O = \Sigma_J^I \cap \Sigma_C^O$, $\Sigma_{\hat{C}}^H = \Sigma_C^{ctrl} \setminus \Sigma_J^I$, $S_{\otimes} = \hat{S}_{\otimes}$ and $\Delta_{\otimes} = \hat{\Delta}_{\otimes}$. Hence we can easily prove this corollary as for Theorem 1 and 2. □

Immediately from Corollary 1 and the proof of Theorem 1, we can prove the equivalence between alternating simulation and the local state space. The following corollary gives the detail.

Corollary 2. *Consider an IA A and a DEC C such that C conforms to A. Let $L_{\otimes}$ be the local state space of C with respect to A. Then for $q \in S_C$, $s \in S_A$, $q \prec s$ if and only if $\langle q, s \rangle \in S_{\otimes}$.*

3.3 Compositional Verification of DEC Networks

A typical component-based design process combines top-down and bottom-up design. IAs are obtained during system decomposition, together with the synchronisation patterns between them. These IAs are then used for developing or selecting suitable DECs (or for further decomposition). These obtained DECs are next composed to form a concrete component-based design, viz. a closed DEC network, where the synchronisation patterns of IAs are reused as the interconnections between DECs. Hence the IAs capture the abstract communication (or interaction) protocols expected by the designer of the DECs.

In this section, the foundation for this approach is laid. In Section 3.3.1 and 3.3.2, networks of both DECs and IAs are defined. Basic properties such as consistency and deadlock freedom for both closed and open DEC networks are then formulated in Section 3.3.3 and 3.3.4. Compositional approaches to verifying these properties are presented next, which utilise the more abstract IA networks to alleviate the state space explosion. Furthermore, a modular method for checking the conformance of open DEC networks with IAs is presented in Section 3.3.4, which maximises the benefit of the above approaches for hierarchical component-based designs. Finally in Section 3.3.5, a compositional approach to the verification of safety properties for DEC networks is proposed, which combines IA networks with component local state information to detect safety violations. As a consequence, the costly construction of global state spaces is avoided.

3.3.1 DEC Networks

Similar to RTS networks, DEC networks are composed of components interacting by means of synchronisation vectors. Differently, designing these networks can be much simpler as these vectors can be specified by the interconnection between input/output ports of the component DECs without worrying about the communicated data values. In defining the networks, we still assume a special symbol ϵ which is not a port of any component DEC.

Definition 21. A *DEC network* is defined by $D = (\alpha, \theta, P, \gamma)$, where

- α is a finite set of *external* ports of the network, consisting of two disjoint sets of input ports α^I and output ports α^O. We let $\alpha^\sharp = \alpha \cup \{\epsilon\}$;
- $\theta \colon \alpha \longrightarrow 2^{\mathcal{V}}$ is a total typing function, mapping each port to a set of values;
- P is a finite set of DECs. We let $\alpha_p^\sharp = \alpha_p \cup \{\epsilon\}$ for all $p \in P$;
- $\gamma \subseteq \alpha^\sharp \times \Pi_{p \in P}\, \alpha_p^\sharp$ is a relation indexed by $\alpha^I \cup \bigcup_{p \in P} \alpha_p^O$.

Similar to RTS networks, we call a DEC network *closed* if $\alpha = \emptyset$, or *open* otherwise. We write $D = (P, \gamma)$ for a closed network D. Also, we call γ a set of *interconnections*. In addition, assuming the same symbol "env", we redefine ρ, η and ι for an interconnection $\mathsf{f} \in \gamma$ in order to formulate the well-formedness rule for DEC networks.

Definition 22. Consider a DEC network $D = (\alpha, \theta, P, \gamma)$ and an interconnection $\mathrm{f} \in \gamma$. Let $\alpha_{\mathsf{env}} = \alpha$, $\alpha^I_{\mathsf{env}} = \alpha^O$, $\alpha^O_{\mathsf{env}} = \alpha^I$, $P' = \{\mathsf{env}\} \cup P$, $l \in P'$, and projections $\pi_l \colon \gamma \longrightarrow \alpha_l$ for all l. Then the *producer* of f, denoted by ρ_{f}, is the unique DEC l such that $\pi_l(\mathrm{f}) \in \alpha^O_l$. $\pi_{\rho_{\mathrm{f}}}(\mathrm{f})$ is called the *producer port* of f. Also, the set of *consumers* of f, denoted by η_{f}, consists of all DECs l such that $\pi_l(\mathrm{f}) \in \alpha^I_l$. For any consumer l, $\pi_l(\mathrm{f})$ is called a *consumer port* of f. In addition, an *idler* of f is a DEC l such that $\pi_l(\mathrm{f}) = \epsilon$. The set of idlers is denoted by ι_{f}.

Definition 23. A DEC network D is called *well-formed* if $\theta_{\rho_{\mathrm{f}}}(\pi_{\rho_{\mathrm{f}}}(\mathrm{f})) \subseteq \theta_l(\pi_l(\mathrm{f}))$ holds for all $\mathrm{f} \in \gamma$, $l \in \eta_{\mathrm{f}}$.

The well-formedness rule requires that the values that can be possibly transmitted are valid (or meaningful) to all the receiving DECs. In other words, only ports of matching kinds can be connected in a DEC network. This thus ensures data type compatibility between ports. In the following, we shall only consider well-formed DEC networks.

An example DEC network is visualized as a block diagram in Figure 3.3, where

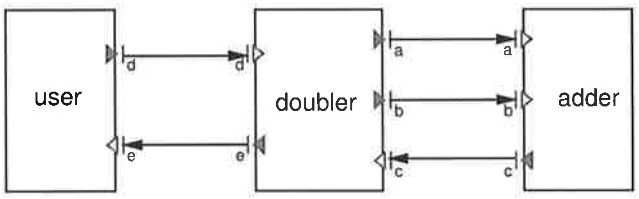
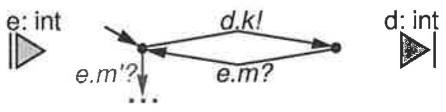


Figure 3.3: A DEC network



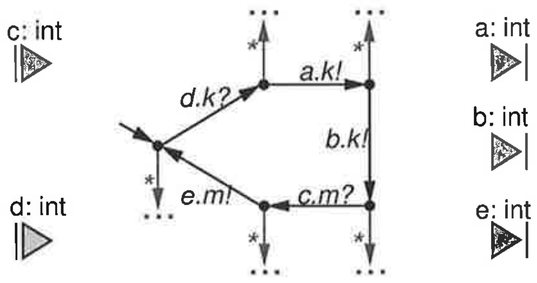


Figure 3.4: A user DEC model

Figure 3.5: A doubler DEC model

rectangles denote DECs and triangles denote DEC ports. This network is closed with no external ports declared. The set P of the network contains the DECs in Figure 3.1, 3.4 and 3.5, where "$*$" matches any unspecified input events. The interconnections γ of the network are depicted by arcs, where ϵ is interpreted as the non-involvement of a DEC in an arc. We know that the adder calculates the sum of two given parameters. The doubler calculates the double of a given number, utilising the function provided by the adder. When receiving a number from port "d", it feeds the adder with the same number at ports "a" and "b", gets the sum at port "c", and then reports back the result via port "e". The user simply uses the service provided by the doubler to calculate the doubles of randomly chosen integers. As such, we have omitted the minutiae for error reporting or recovery.

It is worth noting that disconnected input and output ports of components in a network are allowed. A disconnected input port will receive no data, while a disconnected output port will discard all data sent to it. To facilitate further description, we give the definition below.

Definition 24. Given a DEC network D and a component $p \in P$, the set of *connected ports* of p is defined by

$$\bar{\alpha}_p = \{f \in \alpha_p \mid \exists \mathbf{f} \in \gamma, \pi_p(\mathbf{f}) = f \wedge \eta_{\mathbf{f}} \neq \emptyset\}.$$

We let $\bar{\alpha}_p^I = \bar{\alpha}_p \cap \alpha_p^I$ denote the set of *connected input ports* and $\bar{\alpha}_p^O = \bar{\alpha}_p \cap \alpha_p^I$ the set of *connected output ports*.

Next, we give the semantics of DEC networks in terms of RTS networks.

Definition 25. Consider a DEC network $D = (\alpha, \theta, P, \gamma)$. Let $P' = \{\mathsf{env}\} \cup P$ and $N = (\Sigma, P, R)$ be a derived RTS network such that

- $\Sigma = \Sigma^I \cup \Sigma^O$, $\Sigma^I = \{\langle f, v\rangle \mid f \in \alpha^I, v \in \theta(f)\}$ and $\Sigma^O \subseteq \{\langle f, v\rangle \mid f \in \alpha^O, v \in \theta(f)\}$;
- $\alpha_{\mathsf{env}} = \alpha$, $\alpha_{\mathsf{env}}^I = \alpha^O$, $\alpha_{\mathsf{env}}^O = \alpha^I$, $\Sigma_{\mathsf{env}} = \Sigma$, $\Sigma_{\mathsf{env}}^I = \Sigma^O$, $\Sigma_{\mathsf{env}}^O = \Sigma^I$ and $\theta_{\mathsf{env}} = \theta$;
- $\Sigma^\sharp = \Sigma \cup \{\epsilon\}$, $\Sigma_{\mathsf{env}}^\sharp = \Sigma^\sharp$, and for all $p \in P$, $\Sigma_p^\sharp = \Sigma_p^{obs} \cup \{\epsilon\}$;

- polymorphic projections π_l with $\pi_l \colon \gamma \longrightarrow \alpha_l$ and $\pi_l \colon (\Sigma^\sharp \times \Pi_{p \in P}\Sigma^\sharp_p) \longrightarrow \Sigma^\sharp_l$ for $l \in P'$;
- $R = \{\mathbf{r} \in \Sigma^\sharp \times \Pi_{p\in P}\Sigma^\sharp_p \mid \exists \mathbf{f} \in \gamma, v \in \theta_{\rho_\mathbf{f}}(\pi_{\rho_\mathbf{f}}(\mathbf{f})),$
  $\forall l \in P', (\pi_l(\mathbf{f}) = \epsilon \wedge \pi_l(\mathbf{r}) = \epsilon) \vee (\pi_l(\mathbf{f}) \neq \epsilon \wedge \pi_l(\mathbf{r}) = \pi_l(\mathbf{f}).v)\}$.

Then the *synchronised product* of D is defined by the synchronised product of N.

Note in the definition of R that the data being communicated, viz. v, ranges over the value set defined for the producer port. Further, the data is left unchanged during the transmission from the producer DEC to the consumer DECs. In other words, a DEC network only relays or broadcasts the data but never modifies it. Note also that if all component DECs do not contain the error state $\perp$, then the synchronised product of the network does not contain $\perp$ in its state space either, since DECs are input-universal. In addition, immediately from Definition 11, 25, and Proposition 1, we can get the following proposition.

Proposition 2. *Given a DEC network $D = (\alpha, \theta, P, \gamma)$. Let $L = (s^0, S, \Sigma, \Delta)$ be the synchronised product of D. Then $(\alpha, \theta, s^0, S, \Sigma, \Delta)$ defines a DEC (called the* composite DEC *of D).*

Usually, there are two possible ways to build a DEC network. The first is to include all DECs in a flat network and define the interconnections between them. The second is to assemble some of the DECs and the interconnections between them into an open DEC network. The latter can then be used as a component for building the larger network. The interpretation of DEC networks in terms of DECs in Proposition 2 enables such a hierarchical design. In other words, a DEC network can be composed of components which are in turn DEC networks. Semantically, a hierarchical network is identical to a flattened network which combines all the components and interconnections of the constituent networks, as defined below.

Definition 26. Consider a DEC network $D_1 = (\alpha_1, \theta_1, P_1, \gamma_1)$ and an open DEC network $D_2 = (\alpha_2, \theta_2, P_2, \gamma_2)$ such that $D_2 \in P_1$, the *flattened network* D of D_1 *by* D_2 is defined by $D = (\alpha_1, \theta_1, P, \gamma)$, where $P'_1 = P_1 \setminus \{D_2\}$, $P = P'_1 \cup P_2$, $F = \alpha^\sharp_1 \times \Pi_{p\in P}\alpha^\sharp_p$, and

$$\begin{aligned}\gamma = \{\mathrm{f} \in F \mid \exists \mathrm{f}_1 \in \gamma_1, \mathrm{f}_2 \in \gamma_2, \pi_{D_2}(\mathrm{f}_1) = \pi_{\mathsf{env}}(\mathrm{f}_2) \wedge \pi_{D_2}(\mathrm{f}_1) \neq \epsilon \\ \wedge \forall p \in P_1', \pi_p(\mathrm{f}) = \pi_p(\mathrm{f}_1) \wedge \forall p \in P_2, \pi_p(\mathrm{f}) = \pi_p(\mathrm{f}_2)\}\end{aligned} \tag{3.7}$$

$$\cup \{\mathrm{f} \in F \mid \exists \mathrm{f}_1 \in \gamma_1, \pi_{D_2}(\mathrm{f}_1) = \epsilon \wedge \forall p \in P_1', \pi_p(\mathrm{f}) = \pi_p(\mathrm{f}_1) \wedge \forall p \in P_2, \pi_p(\mathrm{f}) = \epsilon\} \tag{3.8}$$

$$\cup \{\mathrm{f} \in F \mid \exists \mathrm{f}_2 \in \gamma_2, \pi_{\mathsf{env}}(\mathrm{f}_2) = \epsilon \wedge \forall p \in P_1', \pi_p(\mathrm{f}) = \epsilon \wedge \forall p \in P_2, \pi_p(\mathrm{f}) = \pi_p(\mathrm{f}_2)\} \tag{3.9}$$

In the definition, the components of both D_1 and D_2 become components of D and the interconnections of both D_1 and D_2 are unified into the interconnections of D. The unification is based on shared ports of D_2. For example, suppose f_1 is an interconnection in D_1 and f_2 is in D_2 such that f_1 and f_2 involve the same port of D_2, *i.e.* $\pi_{D_2}(\mathrm{f}_1) = \pi_{\mathsf{env}}(\mathrm{f}_2)$. We unify f_1 and f_2 into one interconnection in D such that both $\pi_{D_2}(\mathrm{f}_1)$ and $\pi_{\mathsf{env}}(\mathrm{f}_2)$ are removed (formula 3.7). On the other hand, interconnections in both D_1 and D_2 which do not involve ports of D_2 are expanded and retained in D (formulae 3.8–3.9). Flattening a hierarchical design recursively, we can ultimately get a DEC network with no networks as subcomponents.

3.3.2 IA Networks

In the section, we introduce the concepts of interface automaton networks and define consistency properties for them, which, roughly speaking, refers to the compatibility of constituent IAs in the networks.

Definition 27. An *IA network* is a closed RTS network $N = (W, R)$, where W is a finite set of IAs.

Definition 28. Consider an IA network $N = (W, R)$. Let $L_N = (s^0, S, \Sigma, \Delta)$ be the synchronised product of N. Then N is *consistent* if $\pi_a(\mathrm{s}) \neq \perp$ for all $\mathrm{s} \in S, a \in W$.

An IA network is a restricted RTS network. Its consistency ensures that the network is free from unspecified reception. More specifically, when a synchronisation occurs as a result of an output produced by a constituent IA, for every consumer IA of the corresponding synchronisation vector, the consumed event must be specified at its current state. That is, for every consumer, there always exists a state to enter after consuming the event. In other words, the error state of the input-universal version of any constituent IA is not reachable in the synchronised product of the IA network.

In order to support the compositional verification of deadlock freedom for component-based systems, we define the live consistency for IA networks for use in combination with the live conformance of components.

Definition 29. Consider a consistent IA network $N = (W, R)$. Let $L_N = (s^0, S, \Sigma, \Delta)$ be the synchronised product of N. Then N is *live-consistent* if for all $\mathrm{s} \in S$, $\exists(\mathrm{s}, e, \mathrm{s}') \in \Delta \vee \forall a \in W, \mathfrak{en}_a^I(\pi_a(\mathrm{s})) = \emptyset$.

The live consistency of an IA network ensures that it is free from deadlock. A deadlock refers to a situation where the network cannot make progress whereas some constituent IA is still expecting input events.

3.3.3 Verification of Basic Properties for Closed DEC Networks

In this section, we concentrate on basic properties of closed DEC networks, such as consistency and deadlock freedom, and their verification.

3.3.3.1 Consistency

As noted previously, the IAs and IA networks of Section 3.3.2 capture the abstract interaction protocols expected by the designer of the DECs. The consistency of a closed DEC network is defined in terms of the freedom from unexpected reception. In other words, every possible trace (or event sequence) of a DEC in the network corresponds to a trace of its associated IA, disregarding the internal events and data values accompanying the input/output events of the DEC.

To present a formal definition of the consistency property, we need to formally specify the association between the specification IAs and the implementation DECs as well as the trace projection from the behaviour of the DEC network to that of the IAs. The association is specified in Definition 30 as a sketching function β, which maps each component DEC to an IA which has been used to guide the development or selection of the DEC. The trace projection is defined by Definition 31.

Definition 30. A closed DEC network $D = (P, \gamma)$ is said to be *sketched by* a total function $\beta\colon P \longrightarrow \mathcal{U}^{ia}$ if for all $(p, a) \in \beta$, $\bar{\alpha}_p^I \subseteq \Sigma_a^I$, $\bar{\alpha}_p^O \subseteq \Sigma_a^O$, and p conforms to a, where $\bar{\alpha}_p$ denotes the connected ports of p in D (Definition 24).

Deriving from the trace projection of DECs (cf. Definition 12), we obtain a mapping from the behaviour of the network to that of the IAs associated with the DECs, as defined below.

Definition 31. Consider a closed DEC network D sketched by β. Let σ be a trace of D from s_D^0, $(p, a) \in \beta$, $\hat{p}$ represent $\hat{p}(\Sigma_a^O)$, $\pi_p(\sigma)\restriction_{\Sigma_{\hat{p}}^{obs}}$ be the observable event sequence of the trace projection of σ on p (Definition 2, 12). Then the *trace projection* $\pi_p(\sigma)\restriction_{\Sigma_a}$ *of* σ *on* a is the sequence of ports of p involved in $\pi_p(\sigma)\restriction_{\Sigma_{\hat{p}}^{obs}}$.

We can now define the freedom of unexpected reception for traces and thereafter the consistency of closed DEC networks.

Definition 32. Let D, β, σ be as in Definition 31. Then trace σ is *free from unexpected reception* (with respect to β) if $\pi_p(\sigma)\restriction_{\Sigma_a}$ is a trace of a from s_a^0 for all $(p, a) \in \beta$. D is called *consistent* (with respect to β) if all traces of D from s_D^0 are free from unexpected reception.

In other words, a closed DEC network is consistent if the environment of every DEC is always helpful and never provides input events of an unexpected kind to the DEC. The environment of a DEC (in the network) refers to the open DEC network composed of all the other DECs. Also, given an IA associated with the DEC, we let an IA event refused at a state of the IA represent an unexpected kind (or type) of input event at a corresponding state of the DEC. Often receiving such an event will force the DEC to execute some error report or recovery tasks and thus prevent it from providing the normal functionality to the system.

From the definition, we can prove an important property for traces free of unexpected reception, given by the following proposition and corollary.

Proposition 3. *Consider a closed DEC network D sketched by β and a trace σ of D from s_D^0 such that σ is free from unexpected reception with respect to β. Let $\mathbf{q} \in S_D$ be a state reachable via σ, $(p, a) \in \beta$, and $s_a \in S_a$ be a state reachable via $\pi_p(\sigma)\restriction_{\Sigma_a}$ from s_a^0. Then s_a is a unique state reachable in a via $\pi_p(\sigma)\restriction_{\Sigma_a}$ and $\pi_p(\mathbf{q}) \prec s_a$.*

Proof. Let $\hat{p}$ represent $\hat{p}(\Sigma_a^O)$, $\xi_p = \pi_p(\sigma)\restriction_{\Sigma_{\hat{p}}^{obs}}$ and $\xi_a = \pi_p(\sigma)\restriction_{\Sigma_a}$ for $(p, a) \in \beta$. Then s_a is the only state reachable via ξ_a in a because $\Sigma_a^H = \emptyset$ and a is deterministic. In addition, we know $|\xi_p| = |\xi_a|$ and that every event f in ξ_a corresponds an event $f.v$ in ξ_p from Def. 31. Therefore, it follows by induction that $\pi_p(\mathbf{q}) \prec s_a$. □

Corollary 3. *Let D, β, σ, $\mathbf{q}$, p, a, s_a be as in Proposition 3. Then $\pi_p(\mathbf{q}) \preceq s_a$ if p live-conforms to a.*

Proof. Immediately from Prop. 3 and Def. 17. □

3.3.3.2 Deadlock Freedom

A closed DEC network is called *deadlocked* if it reaches a state where no component can make progress, generally because each is blocked waiting for an input from others, while the event cannot occur. Deadlock freedom usually refers to the ability of the network to make progress or perform computations. In this context, thanks to the interaction protocols captured by IAs, we are able to distinguish deadlocks from normal termination. Normal termination refers to the fact that no component expects any input in a blocked network. Deadlock indicates the existence of such a component. The input expectation of a component is described by the existence of enabled input events in its corresponding IA. In one word, we consider the deadlock freedom of a DEC network to be the absence of deadlocks in its synchronised product. A formal definition is given by the following.

Definition 33. Consider a closed DEC network D which is consistent with respect to β. Let σ be a trace of L_D from s_D^0, $\mathbf{q} \in S_D$ be a reachable state via σ, and $s_a \in S_a$ be the reachable state via $\pi_p(\sigma)\restriction_{\Sigma_a}$ for all $(p, a) \in \beta$. $\mathbf{q}$ is a *deadlock state* if $\mathbf{q}$ is a terminal state in L_D and $\exists a \in \mathsf{image}(\beta), \mathfrak{en}^I_a(s_a) \neq \emptyset$. D is *free from deadlock* (with respect to β) if no deadlock state is reachable in L_D via any trace.

3.3.3.3 Verifying Basic Properties

In order to alleviate the state space explosion problem, a DEC network is not verified directly in this approach. Instead, we utilise the interface automata, the interaction protocol specifications of the DECs, to build an IA network. In the IA network, the IA events are related in the same way as the component ports are interconnected in the DEC network. Based on the IA network, we then determine the basic properties of the DEC network. A formal definition of such IA networks is given below.

Definition 34. Consider a closed DEC network $D = (P, \gamma)$ sketched by β. Let $N = (\mathsf{image}(\beta), \gamma)$. Then N is called the *derived IA network* of D (with respect to β).

Because N reuses the interconnections of D, we can prove an important property for N using the following proposition, assuming that D is consistent.

Proposition 4. *Consider a closed DEC network $D = (P, \gamma)$ which is consistent with respect to β. Let $N = (W, R)$ be the derived IA network of D. Then for all $\mathrm{q} \in S_D$, there exists a unique state $\mathrm{s} \in S_N$ such that $\forall (p, a) \in \beta$, $\pi_a(\mathrm{s})$ is the state reachable in a via $\pi_p(\sigma) \upharpoonright_{\Sigma_a}$.*

Proof. We prove this proposition by induction. Firstly, when $\mathrm{q} = s^0_D$, let $\mathrm{s} = s^0_N$, then we know this proposition holds. Next, suppose this proposition holds on a state $\mathrm{q} \in S_D$ reachable in D via a trace σ. Let $\mathrm{s} \in S_N$ be the state satisfying this proposition for q, $\hat{p}$ represent $\hat{p}(\Sigma^O_a)$, and ξ_a represent $\pi_p(\sigma) \upharpoonright_{\Sigma_a}$ for $(p, a) \in \beta$. Since D is consistent, we can get $\forall (p, a) \in \beta$, $\pi_p(\mathrm{q}) \prec \pi_a(\mathrm{s})$ from Prop. 3. For any step $\mathrm{q} \xrightarrow{e}_D \mathrm{q}'$, we shall prove this proposition holds on q'. Let $\sigma' = \sigma \cdot e$, then

- if $\exists p \in P, e \in \Sigma^H_{\hat{p}}$, then $\pi_p(\sigma') \upharpoonright_{\Sigma_a} = \xi_a$ for all $a \in W$. Clearly, s is the state for which this proposition holds on q'.
- if $\exists p \in P, e \in \Sigma^O_{\hat{p}}$, we know $\pi_p(\mathrm{q}) \xrightarrow{e}_p \pi_p(\mathrm{q}')$. Let $e = f.v$ and $a = \beta(p)$, then since $\pi_p(\mathrm{q}) \prec \pi_a(\mathrm{s})$, we know $\exists (\pi_a(\mathrm{s}), f, u) \in \Delta_a$. Thus $\exists (\mathrm{s}, f, \mathrm{s}') \in \Delta_N$. It is easy to prove that $\pi_a(\mathrm{s}')$ is the state reachable via $\pi_p(\sigma') \upharpoonright_{\Sigma_a}$ for all $(p, a) \in \beta$, since a is deterministic.

In addition, the uniqueness of s is ensured by the uniqueness of the states reachable in a via ξ_a for all $a \in W$. Therefore, this proposition holds. $\square$

The following theorem shows that, to verify the consistency of a closed DEC network, it is sufficient to prove both the conformance of every DEC with its corresponding IA and the consistency of the derived IA network.

Theorem 3. *Consider a closed DEC network $D = (P, \gamma)$ sketched by β. Let $N = (W, R)$ be the derived IA network of D. Then D is consistent with respect to β if N is consistent.*

Proof. We prove this theorem by induction on the length of a trace σ of D from s_D^0. Let $\mathrm{q} \in S_D$ be a state reachable via σ, $\hat{p}$ represent $\hat{p}(\Sigma_a^O)$, and $\xi_a = \pi_p(\sigma) \upharpoonright_{\Sigma_a}$ be the trace projection of σ on a for $(p, a) \in \beta$ (cf. Def. 31). Then at each step of the induction, we prove that

⟨i⟩. σ is free from unexpected reception;

⟨ii⟩. $\exists \mathrm{s} \in S_N$ such that $\forall a \in W$, $\pi_a(\mathrm{s})$ is the reachable state in a via ξ_a;

Firstly, when $\sigma = \lambda$, we know $\mathrm{q} = s_D^0$. Clearly, ⟨i⟩ holds. Also, $\mathrm{s} = s_N^0$ satisfies ⟨ii⟩ . Next, suppose ⟨i⟩–⟨ii⟩ hold on an arbitrary trace σ of D from s_D^0. For any step $\mathrm{q} \xrightarrow{e}_D \mathrm{q}'$, let a trace $\sigma' = \sigma \cdot e$, then we shall prove ⟨i⟩–⟨ii⟩ hold on σ'. Let $\mathrm{s} \in S_N$ be the state satisfying ⟨ii⟩ for σ and ξ_a' be defined over σ'. Then from Prop. 3 we can get $\forall (p, a) \in \beta$, $\pi_p(\mathrm{q}) \prec \pi_a(\mathrm{s})$.

- if $\exists p \in P$ such that $e \in \Sigma_{\hat{p}}^H$, then $\xi_a' = \xi_a$ for all $a \in W$ and thus ⟨i⟩ holds. Clearly, s is the state for which ⟨ii⟩ holds on σ'.
- if $\exists p \in P$ such that $e \in \Sigma_{\hat{p}}^O$, we know $\pi_p(\mathrm{q}) \xrightarrow{e}_p \pi_p(\mathrm{q}')$. Let $e = f.v$, then $\exists s_a' \in S_a$, $\pi_a(\mathrm{s}) \xrightarrow{f}_a s_a' \wedge \pi_p(\mathrm{q}') \prec s_a'$ (because $\pi_p(\mathrm{q}) \prec \pi_a(\mathrm{s})$). Thus $\xi_a' = \xi_a \cdot f$ is a trace of a.
  - For any DEC $g \in P \setminus \{p\}$ such that $\exists \mathrm{f} \in \gamma, \pi_g(\mathrm{f}) \in \alpha_g^I$, let $f' = \pi_g(\mathrm{f})$, then we have $\exists \pi_g(\mathrm{q}) \xrightarrow{f'.v}_g \pi_g(\mathrm{q}')$. Let $b = \beta(g)$, then since $f' \in \bar{\alpha}_g^I \wedge \bar{\alpha}_g^I \subseteq \Sigma_b^I$, we know $f' \in \Sigma_b^I$. Because N is consistent, no error state exists in L_N and thus $f' \in \mathfrak{en}_b^I(\pi_b(\mathrm{s}))$. Since $\pi_g(\mathrm{q}) \prec \pi_b(\mathrm{s})$, we know $\exists s_b' \in S_b$ such that $\pi_b(\mathrm{s}) \xrightarrow{f'}_b s_b' \wedge \pi_g(\mathrm{q}') \prec s_b'$. Hence $\xi_b' = \xi_b \cdot f'$ is a trace of b.
  - For any other DEC $g \neq p$, let $b = \beta(g)$ and $s_b' = \pi_b(\mathrm{s})$, then $\xi_b' = \xi_b$ is a trace of b.

To sum up, σ' is free from unexpected reception and s_a' is reachable via ξ_a' in a for all $a \in W$. Let s' be the state such that $\pi_a(\mathrm{s}') = s_a'$, then $\mathrm{s}' \in S_N$ and s' satisfies ⟨ii⟩ .

Therefore, this theorem holds. $\square$

The following theorem further shows that, to verify the deadlock freedom of a closed DEC network, it is sufficient to prove both the live conformance of every DEC with its corresponding IA and the live consistency of the derived IA network.

Theorem 4. *Let D, β, N, W and R be as in Theorem 3. Then D is free from deadlock with respect to β if N is live-consistent and for all $(p,a) \in \beta$, p live-conforms to a.*

Proof. From Thm. 3, we know D is consistent. Let σ be a trace of D from s_D^0, $\mathrm{q} \in S_D$ be a state reachable via σ, $\hat{p}$ represent $\hat{p}(\Sigma_a^O)$ for $(p,a) \in \beta$, and $\mathrm{s} \in S_N$ such that $\forall (p,a) \in \beta$, $\pi_a(\mathrm{s})$ is the reachable state via $\pi_p(\sigma)\lceil_{\Sigma_a}$ in a (due to Prop. 4). Then from Cor. 3 we have $\forall (p,a) \in \beta$, $\pi_p(\mathrm{q}) \preceq \pi_a(\mathrm{s})$.

Suppose q is a terminal state, *i.e.* $\forall p \in P, \nexists e \in \Sigma_p^{ctrl}, (\pi_p(\mathrm{q}), e, q) \in \Delta_p$. Hence $\mathfrak{en}_{\hat{p}}^O(\pi_p(\mathrm{q})) = \emptyset$ for all $p \in P$. Because $\pi_p(\mathrm{q}) \preceq \pi_a(\mathrm{s})$, we have $\mathfrak{en}_a^O(\pi_a(\mathrm{s})) = \emptyset$ for all $a \in W$ and thus s is a terminal state. Because N is live-consistent and $\mathrm{s} \in S_N$, we know s is not a deadlock state. Hence $\mathfrak{en}_a^I(\pi_a(\mathrm{s})) = \emptyset$ for all $a \in W$. Therefore, this theorem holds. □

Suppose IAs in Figure 3.2 (page 38), 3.6 and 3.7 have guided us to select DECs in Figure 3.1 (page 36), 3.4 (page 46) and 3.5 (page 46), respectively, in order to compose the DEC network in Figure 3.3 (page 46). To prove the consistency (and deadlock freedom) of this network, we can independently prove the conformance (and live conformance) of the DECs to their corresponding IAs and the consistency (and live consistency) of an IA network consisting of the three IAs. It is not hard to construct the synchronised product and prove the consistency (or live consistency) of the IA network. The product is a RTS shown in Figure 3.8, where all events are internal and no error state exists.

Since IAs capture the interface behaviour of DECs, the IA network describes a superset of the interaction patterns between DECs, which involves neither data values

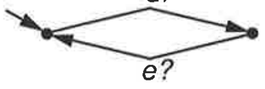


Figure 3.6: A user IA

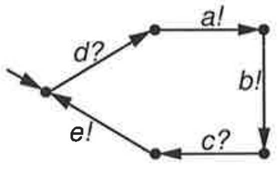


Figure 3.7: A doubler IA

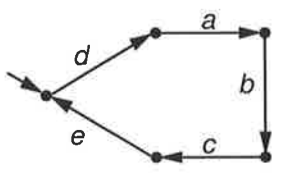


Figure 3.8: Synchronised product of the IA network

nor internal computations of DECs. As a consequence, IA networks generally have smaller state spaces than DEC networks. Hence it is much cheaper to determine the (live) consistency of IA networks. Further, because conformance checking involves only a single DEC at a time, the state space that needs to be handled is also much smaller. Therefore, using this divide-and-conquer approach, the potential state space explosion in the verification of the basic properties of DEC networks can be alleviated.

3.3.4 Verification of Basic Properties for Open DEC Networks

In order to enable verification at each level of the hierarchy, we present compositional methods for verifying open DEC networks in this section. The properties under consideration include consistency, deadlock freedom and conformance.

3.3.4.1 Consistency

To define the consistency for open DEC networks, we let "env" be a syntactical term referring to the environment of a network and associate env with an IA in defining the sketching function. The IA captures the interaction protocol expected of any DEC (or DEC network) later acting as the environment of the network.

Definition 35. An open DEC network $D = (\alpha, \theta, P, \gamma)$ is said to be *sketched by* a total function $\beta\colon (P \cup \mathsf{env}) \longrightarrow \mathcal{U}^{ia}$ if

- for $(\mathsf{env}, M) \in \beta$, $\Sigma^O_M \subseteq \alpha^I_D$;
- for all $(p, a) \in \beta$, $p \neq \mathsf{env}$ implies $\bar{\alpha}^I_p \subseteq \Sigma^I_a$, $\bar{\alpha}^O_p \subseteq \Sigma^O_a$ and p conforms to a, where $\bar{\alpha}_p$ denotes the connected ports of p in D (Definition 24).

The following definition declares the concept of completed networks of open DEC networks.

Definition 36. Consider an open DEC network $D = (\alpha, \theta, P, \gamma)$ sketched by β. Let $M = \beta(\mathsf{env})$ and any DEC $C \notin P$ such that $\alpha_C^I \cap \alpha_D^O \subseteq \Sigma_M^I$, $\alpha_C^O \cap \alpha_D^I \subseteq \Sigma_M^O$ and C conforms to M. Also, let $P^\circ = P \cup \{C\}$ and

$$\begin{aligned}\gamma^\circ = &(\gamma \cap \Pi_{p \in P^\circ} \alpha_p^\sharp) \\ &\cup \{\mathbf{f}^\circ \in \Pi_{p \in P^\circ} \alpha_p^\sharp \mid \exists \mathbf{f} \in \gamma, \pi_{\mathbf{f}}(D) \in \alpha_D^O \setminus \alpha_C, \pi_{\mathbf{f}^\circ}(C) = \epsilon \wedge (\forall p \in P, \pi_{\mathbf{f}^\circ}(p) = \pi_{\mathbf{f}}(p))\}.\end{aligned}$$

Then the closed DEC network $D^\circ = \{P^\circ, \gamma^\circ\}$ is called the *completed network* of D by C (with respect to β), and C is called the *supplementary DEC to D in* D°.

Basically, a completed network of an open DEC network D contains an additional DEC C that can be arbitrary but still conforms to the IA associated with env. The requirements on ports of C, viz. $\alpha_C^I \cap \alpha_D^O \subseteq \Sigma_M^I$ and $\alpha_C^O \cap \alpha_D^I \subseteq \Sigma_M^O$, ensure that M is able to capture the interface behaviour of C on all its connected ports in D°. In other words, we require $\bar{\alpha}_C^I \subseteq \Sigma_M^I$ and $\bar{\alpha}_C^O \subseteq \Sigma_M^O$, where $\bar{\alpha}_C$ denotes the connected ports of C in D°. Furthermore, the interconnections of D are retained in the completed network except those involving a port in $\alpha_D \setminus \alpha_C$. The interconnections in γ involving an input port in $\alpha_D^I \setminus \alpha_C$ are removed and those involving an output port in $\alpha_D^O \setminus \alpha_C$ are modified, each corresponding to an interconnection $\mathbf{f}^\circ$ in γ° such that $\pi_{\mathbf{f}^\circ}(C) = \epsilon$. In other words, the extra ports of D will be disconnected in the completed network. Semantically, this means that the corresponding input steps of D will be disabled, while the corresponding output steps of D will be executed independently of C.

The following proposition demonstrates that any completed network of D is sketched by a function derived from β.

Proposition 5. *Let D, β, C, D° be as in Definition 36 and* $\beta^\circ = \{C \mapsto \beta(\mathsf{env})\} \cup \{(p, a) \in \beta \mid p \neq \mathsf{env}\}$. *Then D° is sketched by β°.*

Proof. Immediately from Def. 30 and 36. □

The consistency of an open DEC network is then defined in terms of the consistency of any of its completed networks as follows.

Definition 37. Consider an open DEC network $D = (\alpha, \theta, P, \gamma)$ sketched by β. Let D° be a completed network of D and β° as in Proposition 5. Then D is called *consistent* (with respect to β) if D° is consistent with respect to β°.

The following proposition shows that the consistency of hierarchical DEC networks, where open networks are composed, can be determined by checking the consistency at each level of the hierarchy. Therefore, this proposition, together with Proposition 7 (described later), provides a foundation for component-based design in that an IA and its mirror can be used as a pair to guide the independent development of components and their composition patterns.

Proposition 6. *Consider a DEC network $D_1 = (\alpha_1, \theta_1, P_1, \gamma_1)$ and an open DEC network $D_2 = (\alpha_2, \theta_2, P_2, \gamma_2)$ such that $D_2 \in P_1$, D_1 and D_2 are consistent with respect to β_1 and β_2, respectively, and $\beta_2(\mathsf{env})$ is the mirror of $\beta_1(D_2)$. The flattened network D of D_1 by D_2 is consistent with respect to β, where*

$$\beta = \{(p, a) \in \beta_1 \mid p \neq D_2\} \cup \{(p, a) \in \beta_2 \mid p \neq \mathsf{env}\}.$$

Proof. Let D° be a completed network of D, C be the supplementary DEC in D°, and β° be as in Prop. 5. Then we need to prove that D° is consistent with respect to β°. From Prop. 5, D° is sketched by β°. We then prove by induction that given a trace σ of D° from $s^0_{D^\circ}$, for all $(p, a) \in \beta^\circ$, $\xi_a = \pi_p(\sigma)\restriction_{\Sigma_a}$ is always a trace of a. We denote this condition as $\langle \mathrm{i} \rangle$.

First, when $\sigma = \lambda$, $\langle \mathrm{i} \rangle$ clearly holds. Next, suppose $\langle \mathrm{i} \rangle$ holds on an arbitrary trace σ of D° from $s^0_{D^\circ}$. Let $A = \beta_1(D_2)$, $M = \beta_2(\mathsf{env})$, and $s_a \in S_a$ be the state reachable via ξ_a for all $a \in \mathsf{image}(\beta) \cup \{A\}$. For any step $\mathrm{q} \xrightarrow{e}_D \mathrm{q}'$, let a trace $\sigma' = \sigma \cdot e$, then we shall prove $\langle \mathrm{i} \rangle$ holds on σ'.

- If $\exists p \in P \cup \{C\}$ such that $e \in \Sigma^H_p$, then $\langle \mathrm{i} \rangle$ holds on σ' for all $(p, a) \in \beta^\circ$;
- If $\exists p \in P_1 \cup \{C\} \setminus \{D_2\}$ such that $e \in \Sigma^O_p$, then $\langle \mathrm{i} \rangle$ holds on σ' for all $(p, a) \in \beta^\circ \setminus \beta_2$, since D_1 is consistent. Let $e = f.v$, $\mathrm{f} \in \gamma_1$ (such that $\pi_p(\mathrm{f}) = f$) and $f' = \pi_{D_2}(\mathrm{f})$, then
  - If $f' = \epsilon$, then $\langle \mathrm{i} \rangle$ holds on σ' for all $(p, a) \in \beta_2$ such that $p \neq \mathsf{env}$;

  - If $f' \in \alpha^I_{D_2}$, we have $f' \in \Sigma^I_A$, since f' is a connected port of D_2 in D_1. Because D_1 is consistent, we know $f' \in \mathfrak{en}^I_A(s_A)$ and thus $f' \in \mathfrak{en}^O_M(s_A)$. Also, because D_2 is consistent with respect to $f'.v$, ⟨i⟩ holds on σ' for all $(p, a) \in \beta_2$ such that $p \neq \mathsf{env}$;

- If $\exists p \in P_2$ such that $e \in \Sigma^O_p$, then ⟨i⟩ holds on σ' for all $(p, a) \in \beta_2$ such that $p \neq \mathsf{env}$, since D_2 is consistent. Also, we know $\exists f.v \in \Sigma^{ctrl}_{D_2}$ such that $(\Pi_{p \in P_2}\pi_p(\mathrm{q}), f.v, \Pi_{p\in P_2}\pi_p(\mathrm{q}')) \in \Delta_{D_2}$. Because $D_2 \in P_1$ and D_1 is consistent, ⟨i⟩ holds on σ' for all $(p, a) \in \beta^\circ \setminus \beta_2$.

Therefore, this proposition holds. □

3.3.4.2 Deadlock Freedom

Similar to the consistency, the deadlock freedom of an open DEC network is defined in terms of its completed network as follows.

Definition 38. Let D, β, C and D° be as in Definition 36 such that C live-conforms to $\beta(\mathsf{env})$ and β° be as in Proposition 5. Then D is *free from deadlock* (with respect to β) if D° is free from deadlock with respect to β°.

Similar to Proposition 6, the following proposition shows that the deadlock freedom of hierarchical DEC networks, where open networks are composed, can be determined by checking the deadlock freedom at each level of the hierarchy.

Proposition 7. *Let D_1, β_1, D_2, β_2, D and β be as in Proposition 6. Then D is free from deadlock with respect to β if D_1 and D_2 are free from deadlock with respect to β_1 and β_2, respectively.*

Proof. Let D° be a completed network of D and β° be as in Prop. 5. Then we need to prove that D° is free from deadlock with respect to β°. Clearly, D° is consistent with respect to β° from Prop. 6.

Let σ be a trace of D° from $s^0_{D^\circ}$, $\mathrm{q} \in S_{D^\circ}$ be a state reachable via σ, and s_a is the reachable state via $\pi_p(\sigma)\upharpoonright_{\Sigma_a}$ in a for all $(p, a) \in \beta^\circ$. Then from Prop. 3 we have $\forall (p, a) \in \beta^\circ$, $\pi_p(\mathrm{q}) \prec s_a$.

Suppose q is a terminal state. Then because D_1 and D_2 are free from deadlock, we have $\forall a \in \mathsf{image}(\beta_1) \cup \mathsf{image}(\beta_2)$, $\mathfrak{en}^I_a(s_a) = \emptyset$. Since $\mathsf{image}(\beta^\circ) \subset \mathsf{image}(\beta_1) \cup \mathsf{image}(\beta_2)$, q is not a deadlock state in D°. Therefore, D° is free from deadlock with respect to β° and thus this proposition holds. □

3.3.4.3 Verifying Consistency

Similar to the consistency checking of closed networks, the consistency of open DEC networks can also be determined by the consistency of their derived IA networks. In the following, we first define derived IA networks for open DEC networks by Definition 39 and then present a compositional method for verifying the consistency of these DEC networks by Theorem 5.

Definition 39. Consider an open DEC network $D = (\alpha, \theta, P, \gamma)$ sketched by β. Let $M = \beta(\mathsf{env}), W = \mathsf{image}(\beta)$ and

$$\begin{aligned} R = {} & (\gamma \cap \Pi_{a \in W} \Sigma_a^{\sharp}) \\ & \cup \{\mathbf{r} \in \Pi_{a \in W} \Sigma_a^{\sharp}) \mid \exists \mathbf{f} \in \gamma, \pi_{\mathbf{f}}(D) \in \alpha_D^O \setminus \Sigma_M, \pi_{\mathbf{r}}(M) = \epsilon \wedge (\forall a \in W \setminus \{M\}, \pi_{\mathbf{r}}(a) = \pi_{\mathbf{f}}(a))\}. \end{aligned}$$

Then $N = (W, R)$ is called the *derived IA network* of D (with respect to β).

The derived IA network consists of the IAs associated with both env and the DECs, and shares the same interconnections with the open DEC network except that those involving a port in $\alpha_D \setminus \Sigma_M$. As in a completed network of the open network, the interconnections involving a port in $\alpha_D^I \setminus \Sigma_M$ are removed, and those involving a port $f \in \alpha_D^O \setminus \Sigma_M$ are modified such that the previous value f is replaced by ϵ.

Theorem 5. *Consider an open DEC network D sketched by β. Let N be the derived IA network of D. Then D is consistent with respect to β if N is consistent.*

Proof. Let D° be a completed network of D and β° be as in Prop. 5. Clearly, D° is a closed DEC network sketched by β° from Prop. 5. Because N is consistent, we know D° is consistent due to Thm. 3. Therefore, D is consistent with respect to β. $\square$

Suppose we have an open DEC network consisting of the adder in Figure 3.1 and the doubler in Figure 3.5, and a function β mapping env to Figure 3.6 and the two DECs to the IAs in Figure 3.2 and 3.7. Then this theorem allows us to prove the consistency of the open network, even though the user DEC is unknown.

3.3.4.4 Verifying Deadlock Freedom

Similar to the checking method of deadlock freedom for closed networks, the deadlock freedom of open DEC networks can also be determined by checking the live consistency of the derived IA networks as well as the live conformance of the components, as demonstrated by the following theorem.

Theorem 6. *Let D, β, and N be as in Theorem 5. Then D is free from deadlock with respect to β if N is live-consistent and for all $p \in P$, p live-conforms to $\beta(p)$.*

Proof. Let C and D° be as in Def. 36 such that C live-conforms to $\beta(\mathsf{env})$, and β° be as in Prop. 5. Clearly, D° is a closed DEC network sketched by β°. Because N is live-consistent and for all $(p, a) \in \beta^\circ$, p live-conforms to a, we know D° is free from deadlock due to Thm. 4. Therefore, D is free from deadlock with respect to β. □

3.3.4.5 Verifying Conformance

As stated previously, an open DEC network forms a DEC. To check the conformance of such a network to an IA, we can certainly use the checking method proposed in Section 3.2.4. However, this may lead to the state space explosion. Instead, we demonstrate using the following two theorems that the conformance and live conformance of the DEC network can be deduced from its derived IA network as well.

Theorem 7. *Let D, β, and N be as in Theorem 5 and A be the mirror of $\beta(\mathsf{env})$. Then D conforms to A if N is consistent.*

Proof. We know from Thm. 5 that D is consistent with respect to β. Let $M = \beta(\mathsf{env})$, $\hat{D}$ represent $\hat{D}(\Sigma_A^O)$, $\hat{p}$ represent $\hat{p}(\Sigma_{\beta(p)}^O)$ for $p \in P$, and

$$\phi = \{(\mathrm{q}, \pi_M(\mathrm{s})) \mid \mathrm{q} \in S_D, \mathrm{s} \in S_N, \forall p \in P, \pi_p(\mathrm{q}) \prec \pi_{\beta(p)}(\mathrm{s})\}.$$

Then we prove by induction that ϕ is an alternating simulation relation between $\hat{D}$ and A (Note $S_M = S_A$). Firstly, $(s_{\hat{D}}^0, s_A^0) \in \phi$ because $\pi_M(s_N^0) = s_M^0 = s_A^0$. Next, suppose $(\mathrm{q}, \pi_M(\mathrm{s})) \in \phi$ for $\mathrm{q} \in S_D, \mathrm{s} \in S_N$,

- For $e \in \Sigma^H_{\hat{D}}$, if $\exists \mathrm{q} \xrightarrow{e}_D \mathrm{q}'$, we prove $(\mathrm{q}', \pi_M(\mathrm{s})) \in \phi$.
    - If $\exists p \in P$ such that $e \in \Sigma^H_{\hat{p}}$, then $(\mathrm{q}', \pi_M(\mathrm{s})) \in \phi$;
    - Otherwise, we have $\exists p \in P$ such that $e \in \Sigma^O_{\hat{p}}$. If $\nexists \mathrm{f} \in \gamma$ such that $p = \rho_{\mathrm{f}} \wedge \eta_{\mathrm{f}} \neq \emptyset$, same as above, we know $(\mathrm{q}', \pi_M(\mathrm{s})) \in \phi$. Otherwise, we have $\exists \mathrm{f} \in \gamma$, $\pi_{\mathsf{env}}(\mathrm{f}) = \epsilon \wedge \pi_p(\mathrm{f}) = f$ such that $e = f.v$. Let $a = \beta(p)$, then since $p \prec a$, we know $f \in \mathfrak{en}^O_a(\pi_a(\mathrm{s}))$. Because N and D are consistent, $\exists \mathrm{s}' \in S_N, \mathrm{s} \xrightarrow{f}_N \mathrm{s}' \wedge \pi_M(\mathrm{s}) = \pi_M(\mathrm{s}') \wedge \forall g \in P, \pi_g(\mathrm{q}') \prec \pi_{\beta(g)}(\mathrm{s}')$. Hence $(\mathrm{q}', \pi_M(\mathrm{s})) \in \phi$;
- For $f.v \in \Sigma^O_{\hat{D}}$, if $\exists \mathrm{q} \xrightarrow{f.v}_D \mathrm{q}'$, then $\exists p \in P, \mathrm{f} \in \gamma, \pi_p(\mathrm{f}) \in \alpha^O_{\hat{p}} \wedge \pi_{\mathsf{env}}(\mathrm{f}) = f$. Let $a = \beta(p)$ and $f' = \pi_p(\mathrm{f})$, then since $p \prec a$, we can get $f' \in \mathfrak{en}^O_a(\pi_a(\mathrm{s}))$ and $\exists \mathrm{s}' \in S_N$, $\mathrm{s} \xrightarrow{f'}_N \mathrm{s}' \wedge \pi_M(\mathrm{s}') \neq \perp$ $\wedge\, \forall g \in P, \pi_g(\mathrm{q}') \prec \pi_{\beta(g)}(\mathrm{s}')$. Hence, $(\mathrm{q}', \pi_M(\mathrm{s}')) \in \phi$;
- For $f \in \Sigma^I_A$, if $\exists \pi_M(\mathrm{s}) \xrightarrow{f}_A s'$ then $f \in \mathfrak{en}^O_M(s)$ and $\pi_M(\mathrm{s}) \xrightarrow{f}_M s'$. Since N is consistent, $\exists \mathrm{s}' \in S_N$, $\pi_M(\mathrm{s}') = s' \wedge \mathrm{s} \xrightarrow{f}_N \mathrm{s}'$. Also, since D is input-universal and $f \in \alpha^I_D$ (Def. 35), $\forall v \in \theta(f)$, $\mathrm{q} \xrightarrow{f.v}_D \mathrm{q}'$. It is easy to prove $\forall p \in P, \pi_p(\mathrm{q}') \prec \pi_{\beta(p)}(\mathrm{s}')$, since D is consistent. Hence $(\mathrm{q}', \pi_M(\mathrm{s}')) \in \phi$;

Therefore, ϕ is an alternating simulation between $\hat{D}$ and A. From Def. 19, D conforms to A. □

Compared with the live consistency, it is much more complicated to deduce the live conformance for open DEC networks, because it involves proving starvation freedom for the environment. Apart from the necessary conditions required for proving the deadlock freedom of these networks, an additional condition is imposed on the formed IA networks to ensure the absence of internal cycles and the possibility of external output events from each state.

Theorem 8. *Let D, β, and $N = (W, R)$ be as in Theorem 5, $M = \beta(\mathsf{env})$, A be the mirror of M, and*

$$E_v = \{e \in \Sigma^O_a \mid a \in W \setminus \{M\}, \exists \mathrm{r} \in R, \pi_a(\mathrm{r}) = e \wedge \pi_M(\mathrm{r}) \in \Sigma_M\},$$
$$E_i = \{e \in \Sigma^O_a \mid a \in W \setminus \{M\}\} \setminus E_v$$

be the sets of output events of other IAs visible and invisible to M, respectively. Then D live-conforms to A if

- *N is live-consistent and for all $p \in P$, p live-conforms to $\beta(p)$;*

- *M is not starved in N,* i.e. *for any state $\mathrm{s} \in S_N$, either $\mathfrak{en}^I_M(\pi_M(\mathrm{s})) = \emptyset$ or the following conditions hold:*

    1. *For all $\mathrm{s}' \in S_N \setminus \{\mathrm{s}\}$ such that s' is reachable from s by E_i, s is not reachable from s' by E_i;*
    2. *There exists a state $\mathrm{s}' \in S_N$ reachable from s by E_i such that $\exists e \in E_v$, $(\mathrm{s}', e, \mathbf{u}) \in \Delta_N$ and $\nexists e' \in E_i$, $(\mathrm{s}', e', \mathbf{u}') \in \Delta_N$.*

Proof. We know from Thm. 7 that D conforms to A. Let $\hat{D}$ represent $\hat{D}(\Sigma^O_A)$, $\hat{p}$ represent $\hat{p}(\Sigma^O_{\beta(p)})$ for $p \in P$, $\mathrm{q} \in S_D$ and $\mathrm{s} \in S_N$ such that $\forall p \in P, \pi_p(\mathrm{q}) \prec \pi_{\beta(p)}(\mathrm{s})$ and thus $\mathrm{q} \prec \pi_M(\mathrm{s})$. Suppose $\mathfrak{en}^O_A(\pi_M(\mathrm{s})) \neq \emptyset$. Then $\mathfrak{en}^I_M(\pi_M(\mathrm{s})) \neq \emptyset$. From Cond. 1 & 2, we know the set of states in N reachable by E_i from s form a tree with each edge labelled by an event in E_i. Because N is finite and these states are not mutually reachable by E_i (due to Cond. 1), every trace by E_i from s to a leaf of this tree is also finite.

⟨i⟩. If $\nexists(\mathrm{s}, e, \mathrm{s}') \in \Delta_N$ for all $e \in E_i$ (*i.e.* this tree has only one node), then $\exists a \in W \setminus \{M\}$, $\mathfrak{en}^O_a(\pi_a(\mathrm{s})) \neq \emptyset$ and $\forall a' \in W \setminus \{M\}$, $\mathfrak{en}^O_{a'}(\pi_{a'}(\mathrm{s})) \subseteq E_v$. Let $p \in P$ such that $(p, a) \in \beta$. Then since p live-conforms to a, we have $\mathfrak{en}^O_{\hat{p}}(\pi_p(\mathrm{q})) \neq \emptyset$ and for all $f.v \in \mathfrak{en}^O_{\hat{p}}(\pi_p(\mathrm{q}))$, $f \in \mathfrak{en}^O_a(\pi_a(\mathrm{s}))$ and thus $f \in E_v$. Hence $\mathfrak{en}^O_{\hat{D}}(\mathrm{q}) \neq \emptyset$.

⟨ii⟩. If $\exists e \in E_i$ such that $(\mathrm{s}, e, \mathrm{s}') \in \Delta_N$, then $\exists a \in W \setminus \{M\}$, $\mathfrak{en}^O_a(\pi_a(\mathrm{s})) \neq \emptyset$. Let $p \in P$ such that $(p, a) \in \beta$. Then since p live-conforms to a, we have $\mathfrak{en}^O_{\hat{p}}(\pi_p(\mathrm{q})) \neq \emptyset$.

- If $\exists f \in E_v$, $v \in \theta_p(f)$, $f.v \in \mathfrak{en}^O_{\hat{p}}(\pi_p(\mathrm{q}))$, then $\mathfrak{en}^O_{\hat{D}}(\mathrm{q}) \neq \emptyset$.
- Otherwise, we have $\forall f.v \in \mathfrak{en}^O_{\hat{p}}(\pi_p(\mathrm{q}))$ such that $f \in E_i$. Let $\mathrm{q}' \in D$ such that $\pi_p(\mathrm{q}) \xRightarrow{f.v}_{\hat{p}} \pi_p(\mathrm{q}')$. Then we know $\exists \mathrm{s}' \in S_N$ such that $\pi_M(\mathrm{s}) = \pi_M(\mathrm{s}')$ and $\forall p \in P, \pi_p(\mathrm{q}') \prec \pi_{\beta(p)}(\mathrm{s}')$. Hence $\mathrm{q}' \prec \pi_M(\mathrm{s}')$ and $\mathfrak{en}^I_M(\pi_M(\mathrm{s}')) \neq \emptyset$. Because the tree from s is finite and s′ is closer than s to some leaves, iteratively applying step ⟨ii⟩, we can ultimately reach a leaf of the tree. For the leaf, step ⟨i⟩ can be applied. In this way, we can prove $\mathfrak{en}^O_{\hat{D}}(\mathrm{q}) \neq \emptyset$.

Therefore, $\mathfrak{en}^O_A(\pi_M(\mathrm{s})) \neq \emptyset$ implies $\mathfrak{en}^O_{\hat{D}}(\mathrm{q}) \neq \emptyset$. We then have $\mathrm{q} \preceq \pi_M(\mathrm{s})$ and hence D live-conforms to A. □

In the theorem, Condition 1 is used to exclude internal cycles in N, where internal events are included in E_i. It basically requires the absence of the internal transitions originating from states which are internally reachable from their target states. Further, Condition 2 states that, if the environment expects an input event (*i.e.* $\mathfrak{en}^I_M(\pi_M(s)) \neq \emptyset$), N is able to produce an expected event after a (possibly empty) internal trace. As will be demonstrated later in Chapter 5, it is easy to check these two conditions using a backward search on S_N.

Applying these theorems into an open DEC network consisting of the adder in Figure 3.1 and the doubler in Figure 3.5, we know this network both conforms and live-conforms to the mirror of the IA in Figure 3.6.

3.3.5 Verifying Safety Properties for DEC networks

Due to the abstraction from DECs to IAs, safety properties of a DEC network cannot be determined merely based on the information contained within the IA network. The information encoded in the local state spaces of the components is also required. In this section, we first define safety properties for DEC networks and then present a compositional approach to their verification.

Definition 40. Given a DEC network $D = (\alpha, \theta, P, \gamma)$, a safety property φ of D is a predicate constraining the possible combinations of values of state variables used by its components.

In the above definition, state variables of a component are referred to as variables that constitute a "system state" of the component and whose value change constitutes a state change of the component. For example, the variable recording the current state of a finite state machine is a state variable. Any change to its value represents a state transition of the machine. Further, a place in a Petri net component is a state variable. A token deposit to it indicates a change in the current state (or marking) of the net. A formal treatment to state variables will be presented later in Chapter 4.

It should be noted that there are generally two kinds of safety properties: *state-based* and *path-based*. In contrast to state-based properties that consider each system state individually as in Definition 40, path-based properties constrain temporal relationships between system states, typically using temporal operators such as "X" (next time), "F" (future time) and "U" (until). Within the context of this thesis, we focus on state-based properties but leave path-based properties for future work. In other words, we say that a closed DEC network satisfies a safety property if the property is evaluated true at every state of the network and the evaluation involves one state at a time.

In the following, we classify safety properties of closed DEC networks into three categories and propose different strategies to check them. The three categories are component local properties, system boundedness and system-wide properties. These properties vary from simple to quite complex ones, and thus the checking methods have different computational efficiency.

3.3.5.1 Local Properties of Components

Local properties of a component are the invariants that the component must maintain during the execution of the composite system.

Definition 41. Given a DEC network $D = (\alpha, \theta, P, \gamma)$, a safety property φ of D is called *local* to a DEC $p \in P$ if φ involves only local state variables of p.

To prove such a property in a closed DEC network, we clarify using the following theorem that only the local state space of a component needs to be further considered.

Theorem 9. *Given a closed DEC network $D = (P, \gamma)$ which is consistent with respect to β, a local safety property φ of $p \in P$ holds on D if φ holds on the local state space of p with respect to $\beta(p)$.*

Proof. Let $a = \beta(p)$ and $L_\otimes = (s^0_\otimes, S_\otimes, \Sigma_\otimes, \Delta_\otimes)$ be the local state space of p with respect to a. Suppose $\mathrm{q} \in S_D$ is a state reachable in D via a trace σ such that q violates φ, *i.e.* $\pi_{\mathrm{q}}(p)$ violates φ. Let $\xi = \pi_p(\sigma)\restriction_{\Sigma_a}$ be the trace projection of σ on a (cf. Def. 31) and $s \in S_a$ be the state reachable in a via ξ. Since σ is free from unexpected reception, we know $\pi_p(\mathrm{q}) \prec s$ from Prop. 3. Hence $\langle \pi_{\mathrm{q}}(p), s \rangle \in S_\otimes$ (due to Cor. 2). Therefore, the violation of φ in D can be detected in the local state space of p, and thus this theorem holds. □

This theorem shows that a safety property local to a DEC is preserved in a consistent network comprising the DEC. Therefore, to prove such a property in the network, it is sufficient to prove it in the local state space of the DEC.

We next generalise this theorem to open DEC networks using the following theorem.

Theorem 10. *Given an open DEC network $D = (\alpha, \theta, P, \gamma)$ which is consistent with respect to β, a local safety property φ of $p \in P$ holds on the local state space of D with respect to the mirror of $\beta(\mathsf{env})$ if φ holds on the local state space of p with respect to $\beta(p)$.*

Proof. It is trivial to prove the local state space of D is a closed DEC network. Then, due to Theorem 9, this theorem clearly holds. □

This theorem justifies that any component invariant involved in a hierarchical DEC network can be proved at the local state space of the component, no matter how deep the component is at the hierarchy of composition.

3.3.5.2 System Boundedness

System boundedness is a special system-wide property, which does not require comprehensive analysis proposed later and is therefore worthwhile mentioning before we continue. For a given closed DEC network, system boundedness refers to the fact that all the internal buffers (or data containers) defined in every component are bounded. In other words, the number of elements stored in any buffer is at all times below a certain bound. This property can be regarded as the conjunction of the boundedness of all its components. The boundedness of a particular DEC indicates the boundedness of all its buffers, which is clearly a local safety property of the DEC. Exceeding any bound of its buffers during the execution of the DEC network is a violation of this local property, and can be detected in the local state space as well, according to Theorem 9. Therefore, system boundedness is deducible from the independent analysis of local state spaces of the components.

3.3.5.3 System-Wide Properties

System-wide properties are the safety properties that are not covered by the above two categories. Generally, they involve local state variables of multiple components. In essence, they can be expressed in conjunctive normal form, with clauses being disjunctions of component local properties. A formal definition of property clauses is given below in Definition 42.

Definition 42. Given a DEC network $D = (\alpha, \theta, P, \gamma)$, a *property clause* φ of D is a formula

$$\varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_k$$

such that $1 \leq k \leq |P|$, φ_j is a local safety property of a component $p_j \in P$ for all $j\colon 1 \leq j \leq k$, and at most one property for each component appears in the formula (*i.e.* $\forall i, j\colon 1 \leq i, j \leq k$, $i \neq j$ implies $p_i \neq p_j$).

Any formula of predicate calculus can be expressed in conjunctive normal form. If a safety property (as in Definition 40) has a logically equivalent formula which is the conjunction of a number of property clauses, then one can independently prove each property clause and accumulatively deduce the property.

Typically, to prove a property clause of a DEC network D, one has to ensure that for every state $\mathrm{q} \in S_D$, φ_j holds at $\pi_{p_j}(\mathrm{q})$ for some $j\colon 1 \leq j \leq k$. As noted previously, this requires the construction of the global state space of D and thus would easily lead to the state space explosion. In the following, we instead present a compositional approach which makes use of the derived IA network to help detect a potential violation of a property clause. To do so, we first define the concept of satisfaction of a local property by an IA state in Definition 43 and then propose the compositional approach based on Theorem 11 and 12 (below).

Definition 43. Consider a DEC C and an IA A such that C conforms to A. Let φ be a local property of C and $L_\otimes$ be the local state space of C with respect to A. A state $s \in S_A$ is said to *satisfy* φ if $\forall \langle q, s \rangle \in S_\otimes$, φ holds on q. The set of states in S_A satisfying φ is denoted as $\mathtt{sat}_\varphi$.

The following theorem studies closed DEC networks and demonstrates that the violation of a property clause at any state q of a closed consistent DEC network D can be detected by the corresponding state s of the derived IA network N (as in Proposition 4). As a consequence, S_N can be used to prove safety properties on S_D.

Theorem 11. *Consider a closed DEC network $D = (P, \gamma)$ which is consistent with respect to β and a property clause φ as in Definition 42. Let $N = (W, R)$ be the derived IA network of D, p_j be the DEC to which φ_j is local and $a_j = \beta(p_j)$ for $j\colon 1 \leq j \leq k$. Then φ holds on D if for all $\mathrm{s} \in S_N$, either $\exists a \in W$, $\pi_a(\mathrm{s}) = \bot$ or $\exists j\colon 1 \leq j \leq k$, $\pi_{a_j}(\mathrm{s})$ satisfies φ_j.*

Proof. Suppose a state $\mathrm{q} \in S_D$ reachable in D via a trace σ such that q violates φ, viz. $\pi_{p_j}(\mathrm{q})$ violates φ_j for all $j\colon 1 \leq j \leq k$. Let $L_{p\otimes} = (s^0_{p\otimes}, S_{p\otimes}, \Sigma_{p\otimes}, \Delta_{p\otimes})$ be the local state space of p with respect to a for $(p, a) \in \beta$, and $\mathrm{s} \in S_N$ such that $\forall (p, a) \in \beta$, $\pi_a(\mathrm{s})$ is the state reachable via $\pi_p(\sigma)\restriction_{\Sigma_a}$. From Prop. 3, we know $\pi_a(\mathrm{s}) \neq \bot \wedge \pi_p(\mathrm{q}) \prec \pi_a(\mathrm{s})$. Hence $\langle \pi_p(\mathrm{q}), \pi_a(\mathrm{s})\rangle \in S_{p\otimes}$ for all $(p, a) \in \beta$ (due to Cor. 2) and $\nexists j\colon 1 \leq j \leq k$ such that $\pi_{a_j}(\mathrm{s})$ satisfies φ_j. Accordingly, s falsifies the condition of this theorem. Therefore, the violation of φ can be detected and thus this theorem holds. □

Note that the consistency of a closed consistent DEC network does not imply the consistency of its derived IA network. Therefore, in the above theorem, the requirement "$\exists a \in W, \pi_a(\mathrm{s}) = \bot$" is imposed to exclude the combinations of IA states to which no state of the DEC network corresponds. In addition, for any other IA state combination, this theorem requires that at least one local property is satisfied.

As noted above, to prove a safety property local to an open DEC network D, it is sufficient to prove it at every state in the local state space of the formed DEC of D. However, this may lead to the state space explosion. Therefore, we demonstrate using the following theorem that to prove a local property clause of D, it is sufficient to prove it at every state of the derived IA network using a method similar to the one in Theorem 11.

Theorem 12. *Consider an open DEC network $D = (\alpha, \theta, P, \gamma)$ which is consistent with respect to β and a property clause φ of D as in Definition 42. Let $N = (W, R)$ be the derived IA network of D, $p_j \in P$ be the DEC to which φ_j is local and $a_j = \beta(p_j)$ for $j\colon 1 \leq j \leq k$. Then φ holds on the local state space of D if for all $\mathrm{s} \in S_N$, either $\exists a \in W$, $\pi_a(\mathrm{s}) = \bot$ or $\exists j\colon 1 \leq j \leq k$, $\pi_{a_j}(\mathrm{s})$ satisfies φ_j.*

Proof. It is trivial to prove the local state space of D is a closed DEC network. Then, due to Theorem 11, this theorem clearly holds. □

This theorem, together with Theorem 11, allows us to separate concerns in proving safety properties of DEC networks and disregard components irrelevant to the verification task. As a consequence, the potential state space explosion problem can be alleviated.

On the basis of Theorem 11 and 12, we propose a compositional method for proving system-wide safety properties. For a given property, we first transform it into conjunctive normal form, *i.e.* the conjunction of property clauses, and then follow a three-step procedure to evaluate each clause φ:

1. calculate the set $\mathtt{sat}_{\varphi_j}$ of IA states satisfying φ_j for all $j: 1 \leq j \leq k$;

2. check every state $s \in S_N$ for the existence of an IA a_j such that $\pi_{a_j}(s) \in \{\perp\} \cup \mathtt{sat}_{\varphi_j}$;

3. if succeed, conclude the satisfaction of φ in D.

Finally, if all constitute clauses of the safety property prove to be true, we conclude that the property holds on D. The implementation of this method will be discussed later in Chapter 5.

3.4 Summary and Discussion

In this chapter, we have presented a practical method for verifying the consistency and deadlock freedom of DEC networks. This employs a divide-and-conquer approach where each component is individually tested for conformance with its interaction protocol captured by an interface automaton, and the network of interface automata (which matches the network of components) is checked for consistency. This divide-and-conquer approach, together with abstraction from the data values transmitted between components, can lead to a significant reduction in the state space to be explored. Furthermore, it was shown that the verification can be carried out at each level of the hierarchy. This further minimises the state space that needs to be built in each verification task.

We have provided only a sufficient condition for determining the consistency and deadlock freedom of DEC networks. This will be appropriate for many practical situations since the consistency of IA networks will be ensured when synchronisation patterns between components are designed during system decomposition, and thus before the development or selection of DECs is carried out. In some cases, however, the use of pre-existing DECs and their abstraction into IAs may mean that the proposed verification process will produce false negatives, i.e. the above technique may incorrectly report inconsistency. In this case, a process of progressive refinement will need to be adopted, as advocated in [35].

Furthermore, we have presented practical and compositional methods for verifying safety properties of consistent DEC networks. Such a verification often requires more information than can be derived merely from the IA network. Therefore, we make use of the component local state spaces to deduce a superset of the state combinations of relevant DECs with the help of the derived IA network, and check this set for the violation of safety properties. In this way, the preservation of such properties can be ensured. In addition, we have proposed methods with different computational efficiency to reason about different kinds of properties, *e.g.* component invariants and system-wide safety properties.

In this chapter, independent analysis (or verification) of components and their composition is strongly advocated, due to the explicit specification of interaction protocols of components. This makes it possible to apply different verification techniques to prove different properties such as the conformance of components, the consistency of IA networks, local and system-wide safety properties. For example, consider a component having infinite data values, then model checking is not directly applicable for ensuring the conformance, whereas theorem proving or the combination of model checking with data abstraction techniques (such as data independence) may be more appropriate. On the other hand, by abstracting away from data values being communicated, IA networks can be verified using model checking tools. In the meantime, other existing techniques attacking the state space explosion may be applied in order to further reduce the size of state space for this verification task.

3.5 Related Work

As noted earlier, interface automata are a key element in this approach. This language was first introduced by de Alfaro and Henzinger in [54]. There, the authors established a simple and well-defined semantics for them and defined the composition of IAs in terms of two-party synchronisation. Also, they proposed alternating simulation as a refinement relationship between IAs. This relationship takes an optimistic view of the environment by assuming that it is always helpful, only supplying inputs expected by the specification automaton. This optimistic view allows more possible implementations than a pessimistic approach where the environment can behave as it pleases.

In this chapter, we have taken this optimistic view and adapted alternating simulation to define the conformance of components with IAs, taking into account data values used in components. Also, we have presented a computationally efficient method of checking this relation, which does not require the construction of the Cartesian product of their state spaces as in [54]. In fact, only the reachable states in the product need to be constructed. Additionally, in contrast to the simple composition scheme in [54], we allow IAs to be composed by means of synchronisation vectors, a more general composition mechanism introduced by Arnold and Nivat [13].

Our adaptation and checking method of alternating simulation was inspired by [173], where a similar relation was proposed to check the conformance (or refinement) relationship between CCS models. However, this relation is more restricted than ours, as it requires both specification and implementation models to have no mixed states (where both input and output transitions can occur), while in our approach this is not required. Furthermore, because of the presence of blocking outputs, models in their approach are different in nature from ours where a component is in full control of its outputs.

As noted in Chapter 2, there exist many other approaches to compositional verification, including the classical assume-guarantee paradigm [81, 171], interface processes [41, 42], and circular reasoning [6, 7, 10, 92, 93, 128, 149, 150, 151]. These approaches attempt to combat the state space explosion using the principle of "divide and conquer". They usually describe the assumptions and guarantees of a component in separate models, and rely on the guarantee models of the other components to validate the assumptions of the component. As the assumptions of components are often interdependent, methods to break the accompanying reasoning circularity have to be developed. Usually the soundness of such methods is ensured by induction on length of traces, *e.g.* [10, 92, 149] .

Our approach has its roots in these approaches. Its underlying methodology was derived from them. However, two key factors make our approach distinct from other compositional verification approaches. Firstly, the temporal relation between the assumed and guaranteed behaviours of a component can be explicitly captured here by a single interface automaton. Accordingly, our approach is able to validate the assumptions of all the components in a step by checking the consistency of the derived IA network. It thus circumvents the common problem of reasoning circularity in assume-guarantee approaches. Secondly, the classical assume-guarantee approaches, *e.g.* [81, 171], take a pessimistic view of the environment. More specifically, to initialize the assume-guarantee chain, one has to ensure the guaranteed properties of at least one component under an unconstrained environment. As noted, this would fail when interdependent assumptions are present. Further, in other approaches, *e.g.* [7, 10, 41, 92, 128, 149, 150], in order to prove the guaranteed properties of a component, a constrained environment model is usually derived from the composition of (abstract) specifications of all the other components. The size of the derived model, nevertheless, may grow very fast with the number of components in the system. In contract, our approach adopts an optimistic view of the environment, which allows one to consider only a component and its associated IA in order to determine its guaranteed properties. It also enables the verification of a hierarchical system at each level of the hierarchy. As a consequence, a more scalable solution is obtained.

The work related to ours also includes compositional minimisation approaches, *e.g.* [32, 71, 78, 79, 193, 200]. These approaches rely on a semantic equivalence to minimise the state spaces of intermediate subsystems, and then use them to generate a reduced but semantically equivalent global state space for verification. However, in these approaches, the verification is only possible when a reduced global state space has been obtained. In contrast, our approach is able to support hierarchical verification and render progressive results. Accordingly, it provides a more effective way to control the complexity of subsystems and avoids the intermediate state space explosion faced by compositional minimisation approaches.

Similar to our approach, the work of [103, 104] also utilises the environmental assumptions of components for verification. In particular, this derives the assumptions and actual behaviours of components from their specifications, and determines the deadlock freedom of a system by pairwise matching between the assumptions of a component and the actual behaviour of another component. In contrast to our approach, the proposed method is incomplete and limited to one-to-one communication or synchronisation.

4
Semantic Interpretation Approach to Heterogeneous Systems

As noted in Chapter 2, the complexity and heterogeneity of modern computer-based systems means that a single specification language is no longer adequate for coping with all aspects of a system. Instead, multiple languages are often used to specify different parts of a system. This results in component-based heterogeneous systems.

For these systems, it is crucial to establish a formal semantics so as to eliminate potential inconsistencies and enable formal verification. To do so, it is essential to develop a common semantic base which is sufficiently general to define the semantics of various modelling languages. It is also important to construct an underlying framework unifying heterogeneous components (*i.e.* components written in different languages) into this semantic base and consequently providing a formal semantics for heterogeneous systems.

In Chapter 3, we have developed a formal semantic base for these systems in terms of interconnected discrete-event components (or reactive transition systems). In this chapter, we focus on a class of modelling languages and study the fundamentals for constructing an underlying framework that provides heterogeneous systems (or components) modelled by these languages with a formal operational semantics based on DECs and their networks.

More specifically, we consider modelling languages that consist of vertices (represented as closed geometrical shapes) and edges (represented as lines) connecting vertices.

This kind of language is often called a *graph-like* notation. Also, we require that the described systems (or components) using such languages have some notion of state, and transit between states due to occurrences of discrete events. Example languages include Petri nets, statecharts and process networks. Our preference for graph-like visual languages results from a previously noted fact that they (or visual languages in general) provide intuitive means for system specification and analysis and are very attractive to software engineers.

In addition, to develop an underlying framework for heterogeneous systems, one may think of building a direct mapping from heterogeneous components into DECs. Although this is theoretically sound, it is generally infeasible to develop automated supporting tools for this approach, because components may have infinite state spaces due to the presence of environment-controlled input events. Also, it is often pointless in practice to construct the whole state space of a component without considering its ultimate context.

Therefore, we do not simply interpret heterogeneous components as DECs. Instead, we define the semantics of a heterogeneous component in terms of its reachable states and transitions within a given context. Also, we observe that the process of computing the reachable states and transitions of a component is often independent of the used modelling language and is a standard element for many analysis tools. Hence, we employ a two-stage process to specify an operational semantics for a component. The first stage deals with the language dependent semantic issues, *e.g.* what constitutes a state or transition of the component, what are the possible transitions (or steps) that the component can make at a particular state, and how the component executes a step according to the semantics of its modelling language. The second stage relies on the facilities developed in the first stage and handles the language independent issues, *e.g.* the computation of the reachable states and the construction or exploration of the state space in the given context.

In this chapter, we concentrate on the first stage, presenting a general approach for graph-like languages to deal with language dependent issues. In Chapter 5, we shall elaborate the second stage, implementing two analysis tools which give an operational semantics to heterogeneous components in terms of DECs, more specifically, in terms of their reachable states and steps within a given context. To enable independent development of these two stages, we impose a contract which explicitly defines the expected behaviours of both the interpreters (see below) defined for the first stage and the analysis tools developed for the second stage.

To deal with the language dependent issues, we adopt the semantic interpretation approach taken by the Moses tool suite [65]. As noted in Chapter 2, such an interpretation approach has advantages over syntactical translation approaches in terms of flexibility and extensibility. More specifically, we use *attributed graphs* [111] to represent the abstract syntax of various component models, and specify an interpreter for each modelling language. When parameterized by the attributed graph of a well-formed component model, the interpreter is able to provide sufficient information about states and transitions of the component, according to the semantics of the language. As a result, an analysis tool can utilise this information to construct the state space of a system comprising the component.

To define language-specific interpreters, as in [65], we employ a variant of *Abstract State Machines* (ASMs) [82], called *Object Mapping Automata* (OMAs) [112], as the description language. ASMs are an expressive and elegant model of computation, combining algebras (first-order structures) and transition systems. They are able to represent static aspects of a component in algebras, such as its data structures and consequently its states, and formalize dynamic aspects of a component (*i.e.* its behaviour) by means of transition systems [56]. Therefore, they are well suited to our needs to characterise states and transitions of heterogeneous components. In this chapter, we employ OMAs in preference to other ASM languages, *e.g.* [82, 25], simply because OMAs are already supported by our implementation framework, *i.e.* the Moses tool suite. However, we believe that the interpretation approach presented here is independent of OMAs and can work with other ASM languages.

As noted above, the work presented in this chapter builds on the previous work of Janneck and Esser [65, 110] on the Moses tool suite. There, the authors have used a similar OMA-based method to define the semantics of graph-like languages. The aim was to support the modelling and simulation of component-based heterogeneous systems. Here, we extend their work to support the formal verification (or exhaustive analysis) of these systems and to accommodate languages with complex semantics, *e.g.* UML statecharts [159]. In particular, we require interpreters to contain certain *analysis variables* to expose the information about states and transitions of components, which is sufficient to support their exhaustive state space exploration.

The remainder of this chapter is structured as follows. In Section 4.1, the preliminary information is introduced, including attributed graphs, OMAs and interfaces to the implementation platform. In Section 4.2, the contract between interpreters and analysis tools is made explicit so that an open framework for both various languages and various analysis techniques can be obtained. After that, the semantic interpretation approach advocated above is illustrated in Section 4.3 and 4.4 with two example languages: compositional Petri nets presented in [113] and statecharts defined in the Unified Modelling Language (UML) [159]. Petri nets are used to illustrate both the analysis variables that interpreters must expose for components and the behavioural obligations that interpreters must fulfill to support the exhaustive state space exploration of components. UML statecharts are used to demonstrate the applicability of this approach to languages with complex semantics. Finally, a summary of this chapter and comparisons with related work are presented in Section 4.5.

It should be noted that although the semantic interpretation approach reported here has its roots in visual languages, we believe, it is also applicable to textual languages that can represent discrete-event systems.

4.1 Preliminaries

As noted above, our semantic interpretation approach works on the abstract syntax of component models, viz. attributed graphs, and defines language-specific interpreters using a formalism called Object Mapping Automata. In this section, we introduce the concept of attributed graphs and give a brief description of OMAs. Furthermore, we clearly define the interfaces to our underlying implementation platform, namely the Moses tool suite, in order to abstract away the implementation issues irrelevant to the topic of this chapter.

4.1.1 Attributed Graphs

An attributed graph [111] is a mathematical structure where vertices, edges, and the graph itself carry named attributes. Assuming a universal set of attribute names $\mathcal{A}$ and a universal set of attribute values $\mathcal{V}$, it is formally defined as follows.

Definition 44. [111] An *attributed graph* is defined by $\mathsf{G} = (\mathsf{V}, \mathsf{E}, \mathsf{src}, \mathsf{dst}, \mathsf{attr})$, where

- V and E are disjoint sets of vertices and edges, respectively.
- $\mathsf{src}, \mathsf{dst} \colon \mathsf{E} \longrightarrow \mathsf{V}$ are total functions mapping an edge to its source and target (or destination) vertices, respectively.
- $\mathsf{attr} \colon (\mathsf{E} \cup \mathsf{V} \cup \{\star\}) \times \mathcal{A} \rightharpoonup \mathcal{V}$ is a partial *attribute function*, where we assume $\star$ is a special symbol representing the graph itself such that $\star \notin \mathsf{V} \cup \mathsf{E}$.

Attributed graphs are sufficiently general to represent the abstract syntax of graph-like component models [111][1]. They thus provide us with a uniform means to access the syntactic information encoded in component models. Therefore, rather than directly dealing with a variety of models, we shall base our semantic interpretation on attributed graphs.

A detailed discussion of the issues concerning the representation of graph-like models by means of attributed graphs is beyond the scope of this thesis but can be found in [111]. Instead, in Section 4.3.1 and 4.4.1 we shall illustrate with example Petri nets and UML statecharts.

For descriptive convenience, we make "attr" a total function by introducing a keyword $\underline{\mathbf{undef}} \notin \mathcal{V}$ and letting $\mathsf{attr}(o, a) = \underline{\mathbf{undef}}$ for all $o \in \mathsf{E} \cup \mathsf{V} \cup \{\star\}$ and $a \in \mathcal{A}$ such that there is no value corresponding to $\langle o, a \rangle$ in attr.

4.1.2 Object Mapping Automata

Object Mapping Automata (OMAs) [112] are a variant of Abstract State Machines (ASMs) [82], with additional syntactic sugar and a simplified notion of states.

[1]Cf. [111] for a detailed discussion of the syntactic representation and well-formedness issues of graph-like models.

Basically, ASMs (and OMAs) are transition systems, used to provide succinct and executable formal specifications for algorithms [82]. That is, they describe how the specified system of an algorithm transits from one state to another in a number of discrete steps. In ASMs (and OMAs), a state is a multi-sorted first-order structure [25], in particular, a set of named relations and functions of arbitrary arity (note that for technical convenience relations are also viewed as functions taking values in $\{\mathbf{true}, \mathbf{false}\}$). A state transition is a modification of the original valuations of these functions (or relations) at any number of points. Such a modification may be a set of updates to the functions, provided these updates are consistent with each other. Here, the consistency means that any two updates do not apply to the same function at the same point.

As an example, consider the OMA in Program 4.1, describing the computation of the factorial function (taken from [119]): A state of this OMA comprises two functions: a nullary function (*i.e.* a variable or *attribute*) called n and a unary function f. In its initial state (specified by **initialize**), n is bound to the value 1 and f is bound to a function that is undefined everywhere except at point 0, where it has the value 1.

Program 4.1: An OMA computing the factorial function

```
function $n$ arity $0$, $f$ arity $1$
initialize :
            $n := 1, f(0) := 1$
rule step :
      $n := n + 1, f(n) := f(n-1) * n$
```

A step of this OMA is specified by the rule "step". In each execution of this rule, the OMA will modify its state by assigning n a new value and modifying f at the point designated by n. In particular, f is changed at point n from being undefined to the value "$n!$", while remaining unchanged at other points. As the OMA iteratively executes rule "step", f will become the factorial function for naturals.

An important feature of ASMs (and OMAs) is that the modifications described by a rule are simultaneous. More specifically, the right-hand side of all assignments are evaluated at the *old* state, and then the assignments are applied simultaneously.

OMAs extend ASMs in a number of ways. The one most pertinent to this work is that OMA rules may be internally iterative, that is, they may describe a state transition that consists of a number of smaller state transitions, separated by a semicolon rather than a comma. For example, the automaton in Program 4.1 could also be written as in Program 4.2 [119].

Program 4.2: Alternative representation of Program 4.1

```
function $n$ arity $0, f$ arity $1$
initialize :
            $n := 1, f(0) := 1$
rule step :
        $f(n) := f(n-1) * n$ ;
        $n := n + 1$
```

Apart from the assignment (":=") and statement composition operators (sequential ";" and parallel ","), OMAs also support the constructs described below, where B, B_1 or B_2 denotes a statement or a statement block, C a boolean predicate, and parts in square brackets are optional.

- **if** C **then** B_1 [**else** B_2] **end** will execute B_1 when condition C holds or execute B_2 otherwise;
- **let** $x = v$ **:** B **end** will execute B while substituting every occurrence of x in B with value v. Note that if v is an expression, it is evaluated before B is executed;
- **do forall** $x \in X$ [**with** C] **:** B **end** will simultaneously execute B for every element x in a set X such that x satisfies C;
- **choose** $x \in X$ [**with** C] **:** B_1 [**else** B_2] **end** will choose any x in X such that x satisfies C and execute B_1 with the chosen x. It will execute B_2 if no such x exists;

- **loop** B **end** will iteratively execute B until a fixpoint is reached, *i.e.* no further update can be made;

- **import** v = **function** $expr$ **:** B **end** will create a fresh function v specified by expression $expr$ and enable the use of v in executing B.

In order to achieve compositionality and hierarchy, both ASMs and OMAs support inter-component communications. To do so, they classify functions into five kinds: *static*, *controlled*, *monitored*, *shared* and *derived*. The first are those whose values never change during any run of OMAs. In contrast, the other four kinds of functions are dynamic with variable values. These kinds differ at which party has permission to update the functions. For a given OMA, controlled functions are only updatable by the OMA, and monitored functions are only updatable by the environment. Shared functions are updatable by both the OMA and the environment, while derived functions are not updatable by either party but are nevertheless dynamic because they are defined in terms of other functions.

The OMA given in Program 4.3 illustrates these five kinds of functions with $Initial$, gcd, $in1$, $mode$ and $double$ being examples of each kind and **mod** the modulus operator. The main job of this OMA is to compute the greatest common divisor gcd of two integer operands $in1$ and $in2$ provided by the environment. The behaviour of the OMA and the environment is coordinated using a shared function $mode$. Initially, $mode = Wait$ and the OMA waits for the environment to intervene, namely, to provide two operands and to change $mode$ to $Initial$. After that, the OMA initializes two auxiliary functions a and b and sets $mode$ to $Compute$. Then the OMA computes gcd by iteratively executing the part of the "step" rule between lines 12–20 until the greatest common divisor is obtained. Finally, it resets $mode$ to $Wait$ to wait for the next request. Note it is assumed that only when $mode = Wait$ can the environment intervene by changing $mode$ to $Initial$ and modifying $in1$ and $in2$.

For descriptive convenience, we shall use "**attribute** x" as a syntactic shortcut for "**function** x **arity** 0". Similarly, we shall use "**set** X" as a shortcut for "**function** X **arity** 1" with a boolean codomain, and use "$X := X + \{x\}$" to denote the addition of an element x into set X instead of the OMA formula "$X(x) :=$ **true**" and "$X := X - \{x\}$" the removal of x from X instead of "$X(x) :=$ **false**". We shall also interchangeably use "+" and "$\cup$" for

Program 4.3: An interactive OMA for the Euclidean algorithm repeatedly computing the greatest common divisor (adapted from [83])

```
**static function** $Initial = 0, Compute = 1, Wait = 2$ ;
**monitored function** $in1, in2$ **arity** 0 ;
**controlled function** $a, b, gcd$ **arity** 0 ;
**shared function** $mode$ **arity** 0 ;
**derived function** $double = gcd * 2$ ;
**initialize** :
        $mode := Wait$
**rule** $step$ :
    **if** $mode = Initial$ **then**
        $a := in1, b := in2, mode := Compute$
    **end**,
    **if** $mode = Compute$ **then**
        **if** $b = 0$ **then**
            $gcd := a, mode := Wait$
        **elseif** $b = 1$ **then**
            $gcd := 1, model := Wait$
        **else**
            $a := b, b := a$ **mod** $b$
        **end**
    **end**
```

set union as well as "−" and "\" for set subtraction. Likewise, we shall use "**relation** Y" as a shortcut for "**function** Y **arity** 2" with a boolean codomain.

4.1.3 Platform Assumptions

In order to specify language-specific interpreters in a general way, we choose to delegate platform-specific features of a diagram, such as the management of diagram-specific variables and operations reading and updating these variables (*e.g.* guard evaluation and action or function execution), to a runtime environment. In doing so, we clearly define the interfaces for requesting the task executions. More specifically, we assume an external function "eval(g, m)" and an external procedure "exec(a, m)" are provided by the underlying platform. Using eval(g, m), a component (or an interpreter) is able to evaluate a transition guard or expression g, depending on a map m which associates a value with each identifier in scope. The eval function must return a boolean value for a guard. Using procedure exec(a, m), a component can execute an action or a sequence of actions a, depending on a map m. An action may be the function specified by a Petri net transition or the effect of a UML statechart transition. The execution of such an action may involve reading and updating variables in m.

The introduction of these interfaces, we believe, simplifies the definition of interpreters and makes it possible to reuse them in any other platform that supports equivalent interfaces.

4.2 Behavioural Contract with Analysis Tools

As stated previously, to define the semantics of a heterogeneous system, language-specific interpreters are parameterized by the constituent visual models and utilised by analysis tools to generate the state space of the system. To enable the independent development of interpreters and analysis tools, a workable behavioural contract between them is needed. In this section, we present such a contract which not only makes explicit the facilities needed for formal verification but also describes how these facilities should be used.

First of all, we require that interpreters contain externally visible *analysis variables*, characterising states and steps of components and providing sufficient information for exploring their reachable states in a given context. These variables include:

- "*_state*": a shared variable representing the current state of a component. This variable needs to be externally writable so that by modifying it an analysis tool can revisit a state of the component in order to execute the steps that are enabled at the state but have not yet been taken;

- "*_nextSteps*": a set of enabled steps of a component at the current state. Note that there is a slight difference between these steps and DEC steps, since the former are actually used by interpreters to identify a particular (DEC) step to execute. The set *_nextSteps* should not be modifiable by analysis tools;

- "*_evtType*" and "*_evtPara*": shared variables representing the event that just occurred to the component, where *_evtType* records its type and *_evtPara* its parameter.
- "*_mode*": a shared variable indicating the mode of an interpreter. This is used to coordinate an analysis tool and the interpreter for modifying other analysis variables. This variable should be one of three predefined constants: *_WAIT*, *_INPUT* and *_FIRE*, representing that the interpreter is waiting for an analysis tool to operate, is receiving an input, or is firing an enabled step, respectively;
- "*_step2take*": a monitored variable representing a step to fire. This step is specified by an analysis tool and must be a currently enabled step in set *_nextSteps*.

Apart from "*_nextSteps*" being a set, we do not make any further assumptions about the data types of these variables but simply treat them as basic data units, when implementing analysis tools later in Chapter 5.

In addition to analysis variables, we also need to specify the behavioural contract between interpreters and analysis tools so that they can collaborate to support exhaustive state space exploration. The expected behaviour of an analysis tool is described as a state-transition graph in Figure 4.1, where S and Δ are sets of the reachable states and steps to be constructed for a given component, respectively. We have abstracted away from the implementation details of the analysis tool, *e.g.* specific exploration strategies such as breadth-first and depth-first.

As shown in the figure, initially, an analysis tool is expected to wait until the mode of an interpreter becomes *_WAIT*. Then it records the initial state of the component and enters the "scheduling" state, where three nondeterministic choices are available. Firstly, the tool may provide an input and change the interpreter to the "*_INPUT*" mode. It then waits until the input is received and records the last step made by the interpreter. Secondly, the tool may choose a step out of *_nextSteps* to request the interpreter to fire. Likewise, it then waits until the firing is finished and records the last step. Finally, the tool may backtrack to an already visited state for executing an enabled next step that is not yet taken.

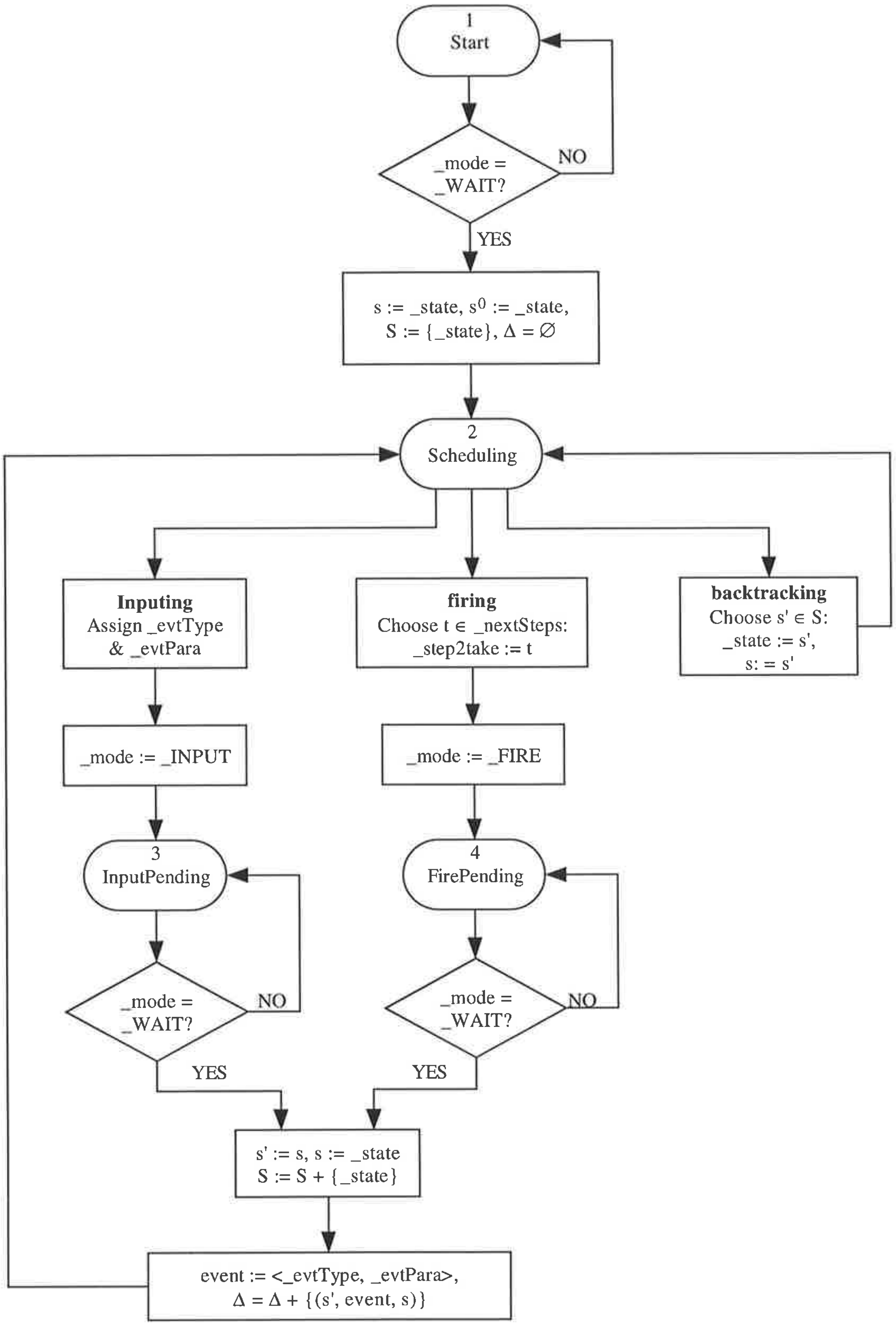


Figure 4.1: Expected behaviour of an analysis tool

With this contract explicitly specified, we are clear about the environment of interpreters. As will be seen, this largely facilitates the interpreter specification in the subsequent sections.

4.3 Petri Nets

In this section, we consider compositional Petri nets introduced by Janneck and Naedele [113], and specify a semantic interpreter for this language. As mentioned previously, our interpreter specification builds on the previous work of [110, 65] and extends the latter with additional facilities to support the exhaustive analysis of Petri net components in heterogeneous systems. More specifically, it contains the analysis variables and abides by the behavioural contract described in Section 4.2.

In the following, we first outline the notation of this language in Section 4.3.1 and illustrate its attributed graphs in Section 4.3.2. We then present our interpreter specification for the language in Section 4.3.3.

4.3.1 Notation

Compositional Petri nets [113] are a variant of high-level Petri nets, which enhances the usual Petri net notation with input/output ports. Through these ports, tokens (or messages) may enter or leave a Petri net component. One may think that, from a net's perspective, an input port represents an external transition and an output port an external place. Due to the incorporation of input/output ports, these Petri nets are also called I/O Petri nets in this thesis.

An example net is shown in Figure 4.2, where triangles represent the input/output ports and where the body of the component is given in the usual Petri Net notation with circles, boxes and arcs representing places, transitions and the flow relationships, respectively. When data (or a token) comes in via an input port, it is added to the place(s) connected to that port. A transition (*e.g.* "*add*" in Figure 4.2) becomes enabled once all its input places have enough tokens and its guard evaluates to true[2]. As is the case for other high-level Petri nets (*e.g.* [116]), this binds the tokens to the variable names (*e.g.* "*va*"

[2]Note that an unspecified guard is always considered to be satisfied.

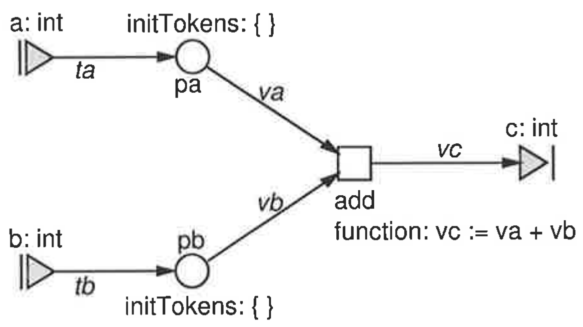


Figure 4.2: An adder Petri net component

and "*vb*") on its incoming arcs, and finally the transition fires. While firing, the transition binds the variable names (*e.g.* "*vc*") on the outgoing arcs depending on the values of the variables on the incoming arcs. When a firing transition is connected to an output port (*e.g.* "*c*"), data is sent out via the port to all connected components.

It is not hard to see that this example net provides the functionality of the adder component in Figure 3.1. Making the same assumptions on the environment, it produces via port c a token with the sum value of two integer tokens coming in from ports a and b. Note that in order to give a concise model, we have again omitted the implementation detail of the "grey" tasks in Figure 3.1.

To obtain a sound semantics, it is important to understand the well-formedness requirements of I/O Petri nets. On the one hand, these nets must meet the requirements of the usual Petri net notation. For instance, Petri nets are bipartite graph, *i.e.* in a Petri net, arcs always connects a place to a transition and vice versa, but are never between places or transitions. On the other hand, due to the introduction of ports, I/O Petri nets have additional constraints on arcs starting from or ending at ports. More specifically, arcs starting from input ports must end at places, and arcs ending at output ports must originate from transitions. Further, we require that at most one output port can be connected to a particular transition in any Petri net, in order to ensure that a Petri net component can exhibit only one output event at a time. Meanwhile, arcs emanating from the same input port and ending at multiple places are allowed, denoting that incoming data are duplicated into connected places. An extensive exposition of the well-formedness requirements will be given later in Section 5.1.

4.3.2 Attributed Graphs

To specify a semantic interpreter for I/O Petri nets, it is important to understand their attributed graphs which, as noted earlier, are the starting point of our approach. In this section, we illustrate using the example net in Figure 4.2. An abridged version of its attributed graph is shown in Program 4.4, where the attributes and values for places, transitions, input and output ports, and edges are illustrated using pa, add, a, c, and va, respectively.

As shown in the program, V and E consist of all vertices and edges in the example net, respectively. Also, src and dst encode the relations between vertices and edges. Compared with them, attr is more complex and requires a more detailed explanation. For a vertex or edge, there is an inherent attribute "*type*" indicating its type. For an I/O Petri net, vertex types include place, transition, input and output ports, while edge types include arcs and inhibitor arcs. This example net has all types of vertices and edges except inhibitor arcs. For different types, the valid attributes are also different. For instance, vertex pa is a

Program 4.4: Attributed graph of Figure 4.2

$\mathsf{V} \equiv \{a, b, pa, pb, add, c\},$

$\mathsf{E} \equiv \{ta, tb, va, vb, vc\},$

$\mathsf{src} \equiv \{ta \mapsto a, tb \mapsto b, va \mapsto pa, vb \mapsto pb, vc \mapsto add\},$

$\mathsf{dst} \equiv \{ta \mapsto pa, tb \mapsto pb, va \mapsto add, vb \mapsto add, vc \mapsto c\},$

$\mathsf{attr} \equiv \{(pa, \text{“}\mathit{type}\text{”}) \mapsto \text{“}\mathit{Place}\text{”}, (pa, \text{“}\mathit{initTokens}\text{”}) \mapsto \text{“}\{\}\text{”},$

$\quad (add, \text{“}\mathit{type}\text{”}) \mapsto \text{“}\mathit{Transition}\text{”}, (add, \text{“}\mathit{function}\text{”}) \mapsto \text{“}\mathit{vc} := \mathit{va} + \mathit{vb}\text{”},$

$\qquad\qquad (add, \text{“}\mathit{guard}\text{”}) \mapsto \underline{\mathbf{undef}},$

$\quad (a, \text{“}\mathit{type}\text{”}) \mapsto \text{“}\mathit{InputPort}\text{”}, (a, \text{“}\mathit{name}\text{”}) \mapsto \text{“}\mathit{a}\text{”}, (a, \text{“}\mathit{domain}\text{”}) \mapsto \text{“}\mathit{int}\text{”},$

$\quad (c, \text{“}\mathit{type}\text{”}) \mapsto \text{“}\mathit{OutputPort}\text{”}, (c, \text{“}\mathit{name}\text{”}) \mapsto \text{“}\mathit{c}\text{”}, (c, \text{“}\mathit{range}\text{”}) \mapsto \text{“}\mathit{int}\text{”},$

$\quad (va, \text{“}\mathit{type}\text{”}) \mapsto \text{“}\mathit{Arc}\text{”}, (va, \text{“}\mathit{label}\text{”}) \mapsto \text{“}\mathit{va}\text{”},$

$\quad \ldots$ /* pb, b, vb, vc, ta, tb *omitted.* */

$\}$

place. It has an attribute "*initTokens*" valued at an empty set, indicating that it is initially empty. Vertex *add* is a transition with attributes "*guard*" and "*function*". The guard states the enabled condition of the transition, and the function specifies the valuation of the variables on the outgoing arcs of the transition. In this case, with an undefined guard representing the truth of its enabled condition, *add* is enabled whenever all its input places have enough tokens. The function of *add* assigns variable *vc* to be the sum of variables *va* and *vb* declared on the incoming arcs of *add*. Further, input/output ports, *e.g.* *a* and *c*, are associated with attributes "*domain*" and "*range*", respectively, to restrict the data flowing through the ports within a valid range. Finally, an arc is associated with an attribute "*label*" to name the token removed from its source place.

It should be noted that in Petri net attributed graphs, attribute values for place initial tokens, transition guards and functions are in fact typed. As we have assumed that handling these values is delegated to the platform, we treat them as uninterpreted strings here for simplicity. Also, since in the attribute function of the example net other vertices and edges are attributed in a similar way, we have omitted them for the sake of brevity.

4.3.3 Semantic Interpretation

As the semantic interpretation issues of general I/O Petri nets have been intensively investigated by Janneck in his PhD thesis [110] in the context of simulation, we do not duplicate his work here. Instead, to emphasize the analysis facilities introduced in our approach, we restrict ourselves to a special class of I/O Petri nets. In particular, we assume in the following that all places in an I/O Petri net are bounded to 1, *i.e.* a place holds at most one token. Thus the marking of a place can be simplified to either the value of the token residing in the place or a special symbol **undef** denoting an empty place. The semantic interpretation approach presented here, nevertheless, can be generalised to all I/O Petri nets (cf. [110] for further details).

Consider a well-formed Petri net model. Let $\mathsf{G} = (\mathsf{V}, \mathsf{E}, \mathsf{src}, \mathsf{dst}, \mathsf{attr})$ be its attributed graph. Then the interpreter of this model is specified in three parts: static function declaration (Program 4.5), analysis variable specification (Program 4.6) and rule definition (Program 4.7).

Program 4.5: Petri net interpreters: static functions

```
static set I = {$v \in \mathsf{V} \mid \mathsf{attr}(v, \text{“}type\text{”}) = \text{“}InputPort\text{”}$},
           O = {$v \in \mathsf{V} \mid \mathsf{attr}(v, \text{“}type\text{”}) = \text{“}OutputPort\text{”}$},
           P = {$v \in \mathsf{V} \mid \mathsf{attr}(v, \text{“}type\text{”}) = \text{“}Place\text{”}$},
           T = {$v \in \mathsf{V} \mid \mathsf{attr}(v, \text{“}type\text{”}) = \text{“}Transition\text{”}$},
           PT = {$e \in \mathsf{E} \mid \mathsf{attr}(\mathsf{src}_e, \text{“}type\text{”}) = \text{“}Place\text{”}, \mathsf{attr}(\mathsf{dst}_e, \text{“}type\text{”}) = \text{“}Transition\text{”}$},
           TP = {$e \in \mathsf{E} \mid \mathsf{attr}(\mathsf{src}_e, \text{“}type\text{”}) = \text{“}Transition\text{”}, \mathsf{attr}(\mathsf{dst}_e, \text{“}type\text{”}) = \text{“}Place\text{”}$},
           IP  = {$e \in \mathsf{E} \mid \mathsf{attr}(\mathsf{src}_e, \text{“}type\text{”}) = \text{“}InputPort\text{”}, \mathsf{attr}(\mathsf{dst}_e, \text{“}type\text{”}) = \text{“}Place\text{”}$},
           TO = {$e \in \mathsf{E} \mid \mathsf{attr}(\mathsf{src}_e, \text{“}type\text{”}) = \text{“}Transition\text{”}, \mathsf{attr}(\mathsf{dst}_e, \text{“}type\text{”}) = \text{“}OutputPort\text{”}$} ;
```

Program 4.5 declares eight static functions as shortcuts for the structural elements of G to facilitate further definition. Among these, I, O, P and T are the sets of input ports, output ports, places and transitions, respectively. Also, PT, TP, IP and TO represent arcs from places to transitions, arcs from transitions to places, arcs from input ports to places, and arcs from transitions to output ports, respectively.

After declaring the static functions, we specify in Program 4.6 the analysis variables required of the interpreter to expose the state and step information of the Petri net. First of all, as a state of a Petri net is a marking of the net, *i.e.* a tuple of markings of places in the net, *_state* is declared as a shared unary function, mapping each place to the value of the token residing in it. In order to be more intuitive, we shall use symbol $\mathcal{M}$ as

Program 4.6: Petri net interpreters: analysis variables

```
shared function _state arity 1 ;                                   /* Also written as “$\mathcal{M}$” */
shared attribute _evtType, _evtPara,
                 _mode ;
derived function buildEnv arity 2
                 = {$(t, \mathsf{attr}(e, \text{“}label\text{”})) \mapsto \mathcal{M}(\mathsf{src}_e) \mid t \in \mathsf{T}, e \in \mathsf{PT}, \mathsf{dst}_e = t$} ;
derived set _nextSteps = {$t \in \mathsf{T} \mid (\nexists e \in \mathsf{PT}, \mathsf{dst}_e = t \wedge \mathcal{M}(\mathsf{src}_e) = $ undef)
                              $\wedge\, \mathsf{eval}(\mathsf{attr}(t, \text{“}guard\text{”}), buildEnv(t))$} ;
monitored attribute _step2take ;
```

an alternative to *_state* later on. Like *_state*, variables *_evtType*, *_evtPara* and *_mode* are also shared between the interpreter and an analysis tool. After that, *_nextSteps* is declared as a derived set consisting of enabled transitions under the current marking $\mathcal{M}$ (or *_state*). A transition t is called *enabled* if all its input places have one token and its guard evaluates to true. Here, the absence of a token in an input place p of t is tested by "$\mathcal{M}(p)$ = **undef**", and the guard evaluation of t relies on an external function "eval". As noted previously, eval is assumed to be provided by the platform, which evaluates an expression depending on given environmental variables. In this case, the guard of t actualises the expression and an ancillary function *buildEnv*(t) provides necessary environmental variables. Here, *buildEnv*(t) is an unary function derived from $\mathcal{M}$, mapping the label of every edge e ending at t into the marking of the source place of e. For instance, consider the net in Figure 4.2. When place *pa* holds a token with value 1 and *pb* holds 2, *buildEnv*(t) will be $\{$"*va*" $\mapsto 1$,"*vb*" $\mapsto 2\}$. Note that as derived functions, *buildEnv* and *_nextSteps* will be re-computed each time $\mathcal{M}$ (or *_state*) is updated. Finally, *_step2take* is declared a monitored attribute since it can only be modified by analysis tools.

In order for the interpreter to work properly, we need to initialize the above-mentioned functions. This process is defined by the initialization rule in Program 4.7. This rule is invoked and executed once when an interpreter is created. As shown in the program, the current marking $\mathcal{M}$ is initialized, depending on the values of attributes "*initTokens*" of places. In the meantime, *_mode* is set to *_WAIT*, which allows an analysis tool to either provide an input or specify an enabled transition to fire.

After the initialization, the interpreter is ready to execute. Its dynamic behaviour is then specified by an iteratively executing "step" rule shown in Program 4.7. In each execution of this rule, two different modes *_INPUT* and *_FIRE* are distinguished. As noted earlier, *_INPUT* means that an input token is ready to be taken, while *_FIRE* states that an enabled transition out of *_nextSteps* is specified for the Petri net to fire. In the former case, we know that *_evtPara* refers to the input token (or message) coming in via an input port of the net and that *_evtType* refers to this input port. As specified in the program, the interpreter handles the input by basically assigning *_evtPara* to the markings of all the places connected with port *_evtType*. In the latter case (*i.e.* when

Program 4.7: Petri net interpreters: rules

```
initialize :
        do forall $p \in \mathsf{P}$ :
                $\mathcal{M}(p) := \mathsf{eval}(\mathsf{attr}(p, \text{"}initTokens\text{"}), \emptyset)$
        end,
        $\_mode := \_WAIT$

rule step :
    if $\_mode = \_INPUT$ then
        do forall $e \in \mathsf{IP}$ with $\mathsf{attr}(\mathsf{src}_e, \text{"}name\text{"}) = \_evtType$ :
                $\mathcal{M}(\mathsf{dst}_e) := \_evtPara$
        end,
        $\_mode := \_WAIT$
    end,
    if $\_mode = \_FIRE$ then
        let $t = \_step2take, outVals = \mathsf{eval}(\mathsf{attr}(t, \text{"}function\text{"}), buildEnv(t))$,
            $P_{in} = \{\mathsf{src}_e \mid e \in \mathsf{PT}, \mathsf{dst}_e = t\}, P_{out} = \{\mathsf{dst}_e \mid e \in \mathsf{TP}, \mathsf{src}_e = t\}$
        :
            do forall $p \in P_{in} \setminus P_{out}$ :
                    $\mathcal{M}(p) :=$ undef
            end,
            do forall $e \in \mathsf{TP}$ with $\mathsf{src}_e = t$ :
                    $\mathcal{M}(\mathsf{dst}_e) := outVals(\mathsf{attr}(e, \text{"}label\text{"}))$
            end,
            choose $e \in \mathsf{TO}$ with $\mathsf{src}_e = t$ :
                    $\_evtType := \mathsf{attr}(\mathsf{dst}_e, \text{"}name\text{"})$,
                    $\_evtPara := outVals(\mathsf{attr}(e, \text{"}label\text{"}))$
            else
                $\_evtType := \tau$,
                $\_evtPara :=$ undef
            end
        end,
        $\_mode := \_WAIT$
    end
```

$\_FIRE$ is flagged), we know that $\_step2take$ refers to the enabled transition to be fired. Let t be an alias to $\_step2take$, then the interpreter fires transition t in three tasks in parallel. Firstly, the markings of the input places of t are cleared (lines 18–20). This, however, excludes the input places that are also the output places of t in order to avoid update clashes with the second task (described below). Here, we let P_{in} and P_{out} be the sets of the input and output places of t, respectively. Secondly, the markings of the output places of t are set (lines 21–23). This involves the use of an auxiliary map $outVals$, associating some outgoing arcs of t with the values determined by the function expression of t. For instance, consider the net in Figure 4.2. When add is firing with $va = 1$ and $vb = 2$, $outVals$ will be $\{\text{“}vc\text{”} \mapsto 3\}$ since $buildEnv(t) = \{\text{“}va\text{”} \mapsto 1, \text{“}vb\text{”} \mapsto 2\}$. Thirdly, a Petri net component may send a message (or data) out via an output port, assuming one is connected to t[3] (lines 24–26). In this case, the name of the connected output port will become the event type $\_evtType$, and the corresponding value in $outVals$ will become the event data $\_evtPara$. For instance, when add fires with $va = 1$ and $vb = 2$ in Figure 4.2, $\_evtType$ will be “c” and $\_evtPara$ will be 3. However, if no output port is connected with the firing transition t (lines 27–29), then we let the event be internal by setting $\_evtType$ to be a special symbol τ, which is not the name of any port, and $\_evtPara$ to be **undef**.

We have now given the interpreter specification for a special class of I/O Petri nets. One can see that the interpreter provides sufficient information about the states and steps of a Petri net, from which an analysis tool can read/modify its current marking, read its currently enabled steps, specify an enabled step to fire, handle its output events, and feed it with input events. As a result, the analysis tool is able to execute the net in a system and explore the state space of the system.

4.4 UML Statecharts

In this section, we shall apply the semantic interpretation approach presented above to a non-trivial language: UML statecharts [159].

The Unified Modelling Language (UML) [159] is a standardised notation for visualising, specifying and documenting object-oriented software systems. UML statecharts (or

[3]Note that using the Petri net well-formedness rules, we have ensured that at most one output port can be connected to a transition.

state diagrams) constitute a principal diagram type in UML for describing the dynamic behaviour of system components. Whereas the syntax and static semantics (or well-formedness) of UML statecharts are relatively precisely defined in UML, their dynamic semantics is only given informally in a natural language and lacks preciseness and completeness. This leaves room for ambiguities and causes problems for formally reasoning about system behaviour and for the development of supporting tools.

In this thesis, we attempt to give them an unambiguous operational semantics, using our semantic interpretation approach. To concentrate the essential ideas conveyed by this chapter, we only consider a subset of this language. This subset, however, does cover core features of UML statecharts such as sequential and concurrent composite states, completion and interlevel transitions as well as variables. A more extensive study on this language can be found in [119].

In the following, we first outline the notation of UML statecharts in Section 4.4.1, illustrate their attributed graphs in Section 4.4.2, and then specify an interpreter for them in Section 4.4.3.

4.4.1 Notation

UML statecharts are an (object-oriented) variant of classical Harel statecharts [86, 87]. The language itself extends traditional state transition diagrams with notions of hierarchy and concurrency. Figure 4.3 shows an example UML statechart diagram, where rectangles represent states, edges represent transitions, and the external events are declared at the top-left corner.

In UML, a statechart diagram is used to model the dynamic behaviour of a class of UML objects. Above all, it specifies states that an object can assume. A state depicts a situation where the object satisfies some condition or waits for some event. States are classified into *simple* states, *sequential composite* states and *concurrent composite* states. Rectangles A, $R1$ and CS in Figure 4.3 are examples of these state classifications, respectively. The hierarchy of a statechart results from the decomposition of a composite state (called the *container*) at a higher level of abstraction into a set of states (called the *substates*), *e.g.* from CS to $R1$ and $R2$, and from $R1$ to A and B. The concurrency of a statechart results from the concurrent threads of control on these substates, *e.g.* $R1$ and

```
InputEvent   a, b;
OutputEvent  c, d;
```

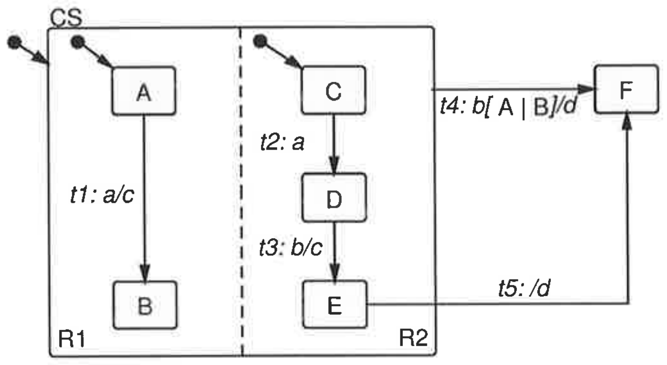


Figure 4.3: An example statechart diagram

$R2$. In this case, the composite state is called a *concurrent* state and the substates are *regions*. A non-concurrent composite state is also called a *sequential* state. Note that regions must be sequential states.

Due to hierarchy and concurrency, an object can assume a set of states at the same time. Such a set specifies a *state configuration* of the object. For instance, $\{\mathsf{top}, CS, R1, A, R2, C\}$ and $\{\mathsf{top}, F\}$ are state configurations of the example statechart. Here, top represents the top state of the statechart. It is the container of all states at the top level and thus exists in every state configuration. States in a state configuration form a tree with the top state at the root and simple states at the leaves, related by the containment (or composition) relation. A state in the current state configuration of the object is said to be *active*.

Furthermore, a statechart also describes the event triggered flow of control of an object due to transition firings which bridge state configurations. Transitions are directed edges between states, *e.g.* $t1, \ldots, t5$ in Figure 4.3, representing complete responses of the object to discrete events. Usually, a transition connects a *source* state and a *target* state, and specifies an event that the object waits for (called the *trigger*), an enabling condition (called the *guard*), and actions to execute (called the *effect*). The last three constitute the transition label, denoted in the form of "trigger[guard]/effect" in Figure 4.3, where unspecified parts are simply omitted. For example, $t1$ has trigger a, no guard, and the effect of generating an event c, and $t4$ has trigger b, guard "$A \mid B$" meaning that A or B is active, and the effect of event d. A transition is called *enabled* and may be fired when the object is in the source state, the trigger event occurs, and the guard is satisfied. Note that an unspecified guard is always considered to be satisfied.

A typical firing of a transition consists of three sequential steps: exiting the source state, executing the transition effect and then entering the target state. Exiting/entering a state will deactivate/activate the state. Additionally, exiting a composite state also involves first exiting all its active substates. Likewise, entering a composite state also involves the subsequent entering of its substate(s). For example in Figure 4.3, when $t4$ fires while A and C are active, the statechart will first exit A, C, $R1$, $R2$ and CS, and then enter F. When exiting, a certain order between these states has to be followed, *i.e.* a substate is always exited first, *e.g.* A prior to $R1$, C prior to $R2$, and both $R1$ and $R2$ prior to CS.

It should be noted that *interlevel transitions*, *e.g.* $t5$, where the source and target states are at different levels of the state hierarchy, are an exception to this firing procedure. Instead, the *main source* is exited in the first step and the *main target* is entered in the last. For instance, the main source and target states of $t5$ in Figure 4.3 are CS and F, respectively. When $t5$ fires, CS will be exited after all active states covered by it. We defer the definitions of the main source and target until Section 4.4.3.4.

In UML, there is a special kind of transition called *completion transitions*, viz. transitions with undefined triggers, *e.g.* $t5$. Such a transition can be enabled by a *completion event* generated for the source state, representing that the source has completed its internal activities. For instance, $t5$ is enabled for firing when E is active. As we do not consider activities within states, we simply regard a state to be completed after it is entered. Also, since we do not consider final states, we only deal with completion transitions emanating from simple states.

In addition, the language of UML statecharts is enriched by the introduction of initial pseudostates. An initial pseudostate, denoted by "•" in Figure 4.3, is used to identify the *default substate* of a sequential composite state, viz. the substate to enter when the composite state is initialized. For example, CS is marked as the default substate of top and will be entered when the statechart is initialized.

The semantics of UML statecharts is based on the *run-to-completion* (RTC) assumption [159]. That is, incoming events are processed one at a time and are dispatched for processing only if the statechart is in a stable state configuration, viz. the processing of the previous event is fully completed [159]. In particular, the events generated in an event processing step are not available in the same step.

When processed, an event may enable multiple transitions. For example, event a enables $t1$ and $t2$ when A and C are active, and event b enables $t3$ and $t4$ when B and D are active. These enabled transitions can be fired simultaneously if the firings do not require a common state to be exited, *e.g.* $t1$ and $t2$. Otherwise, they are called *in conflict*, *e.g.* $t3$ and $t4$. In this thesis, for simplicity, we assume that such conflicts can always be resolved using the lower-first priority [159] which, roughly speaking, gives priority to the transition whose source state is at a lower level of the state hierarchy. For instance, $t3$ will be fired rather than $t4$ if both are enabled. In some cases, an event being processed does not enable any transition. Then the event will be discarded in this context since we do not consider deferrable events.

Let us look at the statechart in Figure 4.3. Initially it is in $\{\mathsf{top}, CS, R1, R2, A, C\}$, as these states are either identified by initial states (*e.g.* CS, A and C) or are regions (*e.g.* $R1$ and $R2$). If sequentially processing events a and b, the statechart will move to $\{\ldots, B, D\}$ by firing both $t1$ and $t2$, and then $\{\ldots, B, E\}$ by firing $t3$. After that, as E is completed, $t5$ will fire and lead the statechart to enter F. The statechart will then stay in F forever since F has no outgoing transitions and hence all events are ignored. On the other hand, if processing b at the initial state configuration, the statechart will directly go to and stay in $\{\mathsf{top}, F\}$.

Next, let us use another example in Figure 4.4 to illustrate the use of variables and internal events (or messages). This figure could be a simple statechart for a refrigerator. The refrigerator consists of a controller, a cooler and a temperature sensor and uses a variable "$temp$" and internal events "$start$" and "$stop$" to coordinate between them. Initially, it is at $\{\mathsf{top}, standby\}$ with temperature $temp = 15$. When switched on, it moves to "working", where "watch", "idle" and "high" are activated. Then $t1$ becomes enabled

```
InputEvent  ON, OFF, FIX;
OutputEvent  ERROR;
VAR  temp = 15;
```

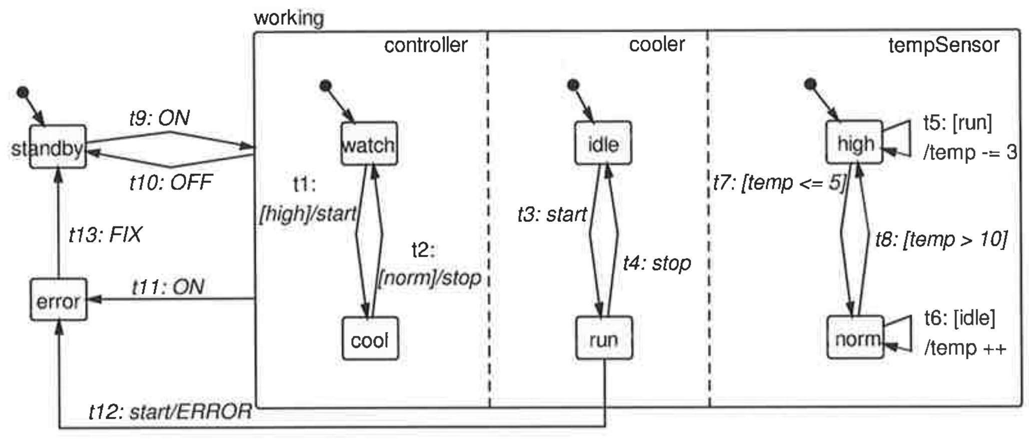


Figure 4.4: A refrigerator statechart

and, when firing, sends a “start” command (or event) which triggers $t3$ at the next step. After $t3$ fires, the refrigerator starts cooling. When the temperature drops below the specified low threshold 5 as a result of repeatedly firing $t5$, the sensor will change to the “norm” state by firing $t7$. This in turn enables $t2$, whose firing will stop the cooler. Then the refrigerator just tries to maintain the temperature. When the temperature slowly rises above the specified high threshold 10 as a result of repeatedly firing $t6$, the statechart will go back to “watch”, “idle” and “high” by firing $t8$. Then a second cycle starts. Here, whenever an event “OFF” is received while the refrigerator is in “working”, $t10$ will fire and lead to the “standby” state. Similarly, $t11$ will fire when receiving “ON” and lead to the “error” state. Analysing this model, it is not hard to see that the trap transition $t12$ for capturing control errors will never be enabled.

Note that this example startchart involves both external (*e.g.* “ON” and “OFF”) and internal events (*e.g.* “start” and “stop”). External events are message exchanges with other components, while internal events are produced and consumed by this component. In this context, events are by default internal unless they are declared by “InputEvent” or “OutputEvent”.

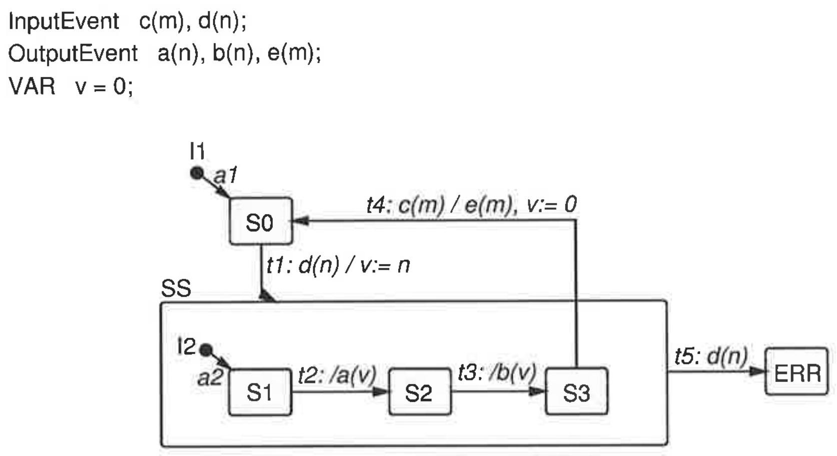


Figure 4.5: A doubler statechart

In order to model real object-oriented systems where inter-object communications often involve parameterized messages, UML statecharts also support parameterized events. Figure 4.5 shows such an example. It depicts a model for the doubler component in Figure 3.5.

At the top-left corner of the figure parameterized input/output events are declared. As described previously, this component can initially take an event $d(n)$ with d the type and n the parameter and set variable v to be n. After that, it will sequentially generate events a and b with v as parameter. If receiving event $c(m)$, it will generate event e with m as parameter and reset v to 0. Note that at states $S1, S2$ and $S3$, d is an unexpected event and its occurrence will cause the "ERR" state to be entered.

Even from these simple examples, one can see that UML statecharts are fairly complicated, involving many features that break the local thread of control. Hence formally defining the language is a nontrivial task. In the following, we concentrate on the above-mentioned features of UML statecharts, although we believe the proposed approach can be generalised to cover the complete language. A more extensive coverage has been reported in [119]. Further, as this research is based the notion of untimed asynchronous communicating components, we only consider signal events in UML, while excluding call events and time events. For descriptive convenience, we shall call an object, whose behaviour is specified by a statechart, an *instance* of the statechart.

4.4.2 Attributed Graphs

As is the case for Petri nets, it is important in our approach to understand the attributed graphs of UML statecharts in order to specify an interpreter for them. In this section, we illustrate with the attribute graph of the doubler statechart in Figure 4.5.

Program 4.8: Attributed graph of Figure 4.5

```
$\mathsf{V} \equiv \{S0, S1, S2, S3, SS, ERR, I1, I2\}$,
$\mathsf{E} \equiv \{t1, t2, t3, t4, t5, a1, a2\}$,
$\mathsf{src} \equiv \{t1 \mapsto S0, t2 \mapsto S1, t3 \mapsto S2, t4 \mapsto S3, t5 \mapsto SS, a1 \mapsto I1, a2 \mapsto I2\}$,
$\mathsf{dst} \equiv \{t1 \mapsto SS, t2 \mapsto S2, t3 \mapsto S3, t4 \mapsto S0, t5 \mapsto ERR, a1 \mapsto S0, a2 \mapsto S1\}$,
$\mathsf{attr} \equiv \{(S0, \text{“type”}) \mapsto \text{“Simple”}, (S0, \text{“name”}) \mapsto \text{“S0”}, (S0, \text{“container”}) \mapsto \underline{\mathbf{undef}}$,
        $(SS, \text{“type”}) \mapsto \text{“Composite”}, (SS, \text{“name”}) \mapsto \text{“SS”}, (SS, \text{“container”}) \mapsto \underline{\mathbf{undef}}$,
                                $(SS, \text{“isConcurrent”}) \mapsto \underline{\mathbf{false}}$,
        $(S1, \text{“type”}) \mapsto \text{“Simple”}, (S1, \text{“name”}) \mapsto \text{“S1”}, (S1, \text{“container”}) \mapsto SS$,
        $(I1, \text{“type”}) \mapsto \text{“Initial”}, (I1, \text{“container”}) \mapsto \underline{\mathbf{undef}}$,

        ...                                          /* S2, S3, ERR, I2 omitted. */

        $(t1, \text{“type”}) \mapsto \text{“Transition”}$,
        $(t1, \text{“trigger”}) \mapsto \text{“d”}$, $(t1, \text{“trgpara”}) \mapsto \text{“n”}$, $(t1, \text{“guard”}) \mapsto \underline{\mathbf{undef}}$,
        $(t1, \text{“effect”}) \mapsto \text{“v := n”}$, $(t1, \text{“sndevt”}) \mapsto \underline{\mathbf{undef}}$, $(t1, \text{“sndpara”}) \mapsto \underline{\mathbf{undef}}$,

        $(t4, \text{“type”}) \mapsto \text{“Transition”}$,
        $(t4, \text{“trigger”}) \mapsto \text{“c”}$, $(t4, \text{“trgpara”}) \mapsto \text{“m”}$, $(t4, \text{“guard”}) \mapsto \underline{\mathbf{undef}}$,
        $(t4, \text{“effect”}) \mapsto \text{“v := 0”}$, $(t4, \text{“sndevt”}) \mapsto \text{“e”}$, $(t4, \text{“sndpara”}) \mapsto \text{“m”}$,

        ...                                          /* t2, t3, t5, a1, a2 omitted. */

        $(\star, \text{“InputEvent”}) \mapsto \{\text{“c”}, \text{“d”}\}$,
        $(\star, \text{“OutputEvent”}) \mapsto \{\text{“a”}, \text{“b”}, \text{“e”}\}$,
        $(\star, \text{“VAR”}) \mapsto \{\text{“v”} \mapsto 0\}$,
        }
```

As shown in Program 4.8, V and E contain all vertices and edges of the doubler statechart, respectively, while src and dst map an edge to its source and target vertices, respectively. The main element in attributed graphs that makes statecharts differ from Petri nets is the attribute function attr. Given a vertex or edge x, attr(x, "*type*") gives the type of x. Given a state s, attr(s, "*name*") and attr(s, "*container*") return its name and the state directly containing s. For a composite state, an additional attribute "*isConcurrent*" is used to distinguish between sequential and concurrent states. For instance, since SS is a sequential state, we have attr(SS, "*isConcurrent*") $=$ **false**. Further, given a transition, attributes "*trigger*" and "*trgpara*" record the type and parameter name of the trigger event, respectively. Attribute "*guard*" refers to the guard of the transition. The effect of the transition is denoted by three attributes with "*effect*" denoting statements for modifying declared variables, and "*sndevt*" and "*sndpara*" describing the parameterized event to be generated. The type of the generated event is represented by "*sndevt*", while the parameter is the value of the expression "*sndpara*". Finally, the graph attributes are given in lines 23-25. Attributes "*InputEvent*" and "*OutputEvent*" are mapped to sets of external input and output events of the statechart, respectively. Also, attribute "*VAR*" is linked to a function mapping the name of each variable into its initial value.

Note that in UML statechart attributed graphs, attribute values for transition guards, effects and event parameters are in fact typed. As is the case for Petri nets, we treat them as uninterpreted strings here and delegate their interpretation to our implementation platform using functions eval and exec.

4.4.3 Semantic Interpretation

In the UML standard [159], the semantics of a statechart diagram is described in terms of the operations of a hypothetical machine. The machine is composed of three key components:

- an *event queue* for holding incoming events, as the environment may provide events faster than a statechart can consume them;
- an *event dispatching processor* that selects and dequeues events for processing, one at a time. It implements the *run-to-completion assumption*;

- an *event processor* that processes each dispatched event. The procedure of dispatching, dequeuing and processing an event is often called a *run-to-completion step*.

The nature of the first two components is left undefined in the standard. In this thesis, we implement the event queue as a FIFO queue and let events dispatched according to the order they arrive. This, however, does not exclude other alternatives, *e.g.* a priority queue or a multiset, which can be accommodated with only minor changes.

Consider a well-formed UML statechart diagram. Let $\mathsf{G} = (\mathsf{V}, \mathsf{E}, \mathsf{src}, \mathsf{dst}, \mathsf{attr})$ be its attributed graph. Then, as usual, we specify the interpreter of a UML statechart in three parts: static function declaration, analysis variable specification and rule definition. The rule definition further includes the initialization rule and a "step" rule for event receiving, dequeuing and processing.

4.4.3.1 Static Function Declaration

We first use Program 4.9 to declare a number of static functions as shortcuts for the structural elements of G. This consists of two parts: vertex and edge classification and state hierarchy decoding.

First of all, vertices and edges are classified in lines 1–10. The input and output ports of the diagram are first obtained from the attributes of G. From the vertices, three kinds of states and one kind of pseudostates are identified according to vertex attribute "*type*". These include simple states S_s, concurrent composite states S_{cc}, sequential composite states S_{sc}, and initial pseudostate P_i. S_{cc} and S_{sc} are all typed "*composite*" but distinguished depending on attribute "*isConcurrent*". Next, composite states are unified into a set S_c and states into a set S. It is worth noting that since the top state top is implicit in every statechart, it is added to S_{sc} and thus S_c and S. Further, the set of transitions T includes all the edges except those originating from initial pseudostates.

Next in lines 12–17, the state hierarchy, graphically represented by the diagram, is decoded. This includes the definitions of five functions or relations over vertices: cntr, subs, default, cover and cover′. "cntr" is a containment function mapping a vertex into the composite state directly enclosing it. A vertex v with "$\mathsf{attr}(s, \text{"}\mathit{container}\text{"}) =$ **undef**" is at the top level and thus mapped to top. Also, for a given composite state c, "$\mathsf{subs}(c)$" refers to the set of (direct) substates of c. We let $\mathsf{subs}(s)$ return an empty

Program 4.9: UML statechart interpreters: static functions

```
static set I = $\mathsf{attr}(\star, \text{"}\mathit{InputEvent}\text{"})$,
           O = $\mathsf{attr}(\star, \text{"}\mathit{OutputEvent}\text{"})$,
           $\mathsf{S}_s = \{v \in \mathsf{V} \mid \mathsf{attr}(v, \text{"}\mathit{type}\text{"}) = \text{"}\mathit{Simple}\text{"}\}$,
           $\mathsf{S}_{cc} = \{v \in \mathsf{V} \mid \mathsf{attr}(v, \text{"}\mathit{type}\text{"}) = \text{"}\mathit{Composite}\text{"} \wedge \mathsf{attr}(v, \text{"}\mathit{isConcurrent}\text{"})\}$,
           $\mathsf{S}_{sc} = \{v \in \mathsf{V} \mid \mathsf{attr}(v, \text{"}\mathit{type}\text{"}) = \text{"}\mathit{Composite}\text{"} \wedge \neg\, \mathsf{attr}(v, \text{"}\mathit{isConcurrent}\text{"})\}$
               $\cup \{\mathsf{top}\}$,
           $\mathsf{S}_c = \mathsf{S}_{cc} \cup \mathsf{S}_{sc}$,
           $\mathsf{S} = \mathsf{S}_c \cup \mathsf{S}_s$,
           $\mathsf{P}_i = \{v \in \mathsf{V} \mid \mathsf{attr}(v, \text{"}\mathit{type}\text{"}) = \text{"}\mathit{Initial}\text{"}\}$,
           $\mathsf{T} = \{e \in \mathsf{E} \mid \mathsf{src}_e \in \mathsf{S} \wedge \mathsf{dst}_e \in \mathsf{S}\}$ ;

static function cntr arity 1 = $\{v \mapsto c \mid v \in \mathsf{S} \cup \mathsf{P}_i, c \in \mathsf{S}_c, \mathsf{attr}(v, \text{"}\mathit{container}\text{"}) = c\}$
                               $\cup \{v \mapsto \mathsf{top} \mid v \in \mathsf{S} \cup \mathsf{P}_i, \mathsf{attr}(v, \text{"}\mathit{container}\text{"}) = \mathbf{undef}\}$ ;
static relation subs arity 2 = $\{(\mathsf{cntr}(s), s) \mid s \in \mathsf{S}, s \neq \mathsf{top}\}$ ;
static function default arity 1 = $\{\mathsf{cntr}(\mathsf{src}_e) \mapsto \mathsf{dst}_e \mid e \in \mathsf{E}, \mathsf{src}_e \in \mathsf{P}_i\}$ ;
static relation cover arity 2 = $\{(c, s) \mid c, s \in \mathsf{S}, c \in \mathsf{cntr}^+(s)\}$ ;
static relation cover' arity 2 = $\mathsf{cover} \cup \{(s, s) \mid s \in \mathsf{S}\}$ ;
```

set for a simple state s. For example in Figure 4.3, we have $\mathsf{subs}(CS) = \{R1, R2\}$, $\mathsf{subs}(R1) = \{A, B\}$ and $\mathsf{subs}(A) = \emptyset$. Further, "default($c$)" indicates the default state of a given state $c \in \mathsf{S}_{sc}$. The function first finds the initial pseudostate directly contained by c and then returns the target state of the edge emanating from the pseudostate. For instance in Figure 4.3, we have $\mathsf{default}(\mathsf{top}) = CS$ and $\mathsf{default}(R1) = A$. Note that the well-formed rules of UML statecharts ensure that the container of an initial pseudostate is a sequential composite state and also that the default state is unique. Additionally, given two states s and c, the boolean function "cover" determines whether c *transitively contains* s, in other words, whether there exists a sequence of composite states $v_1, \ldots, v_k \in \mathsf{S}_c$ for $k \leq |\mathsf{S}_c|$ such that $\mathsf{cntr}(s) = v_1$, $\mathsf{cntr}(v_1) = v_2$, ..., $\mathsf{cntr}(v_k) = c$. Here, cntr^+ denotes the transitive closure of cntr. Clearly, top covers all states in the diagram. For example in Figure 4.3, $(CS, R1), (CS, A), (\mathsf{top}, A) \in \mathsf{cover}$. We further let cover′ include an additional identity relation over states. One may think that cover′ corresponds to the reflexive and transitive closure cntr^* of cntr in the same way as cover corresponds to cntr^+. For descriptive convenience, in the following, we shall use $\mathsf{cover}(c, s)$ as a shorthand for "$(c, s) \in \mathsf{cover}$" and $\mathsf{cover}(c)$ for "$\{s \mid (c, s) \in \mathsf{cover}\}$", viz. the set of states covered by $c \in \mathsf{S}$. Similar abbreviations also apply to cover'.

4.4.3.2 Analysis Variable Specification

To understand the behaviour of a UML statechart, it is essential to understand its current "system state". This state is a tuple consisting of not only the current state configuration and the current content of the event queue, but also the valuation of the declared variables in the statechart. However, for descriptive convenience, we declare it as three functions in Program 4.10, with Δ denoting the current state configuration, Q_{evt} the event queue and $vars$ a function from the declared variable names to their values. Also, we assume simultaneous reading and writing on these three functions by an analysis tool. That is, reading the current "system state" will read them simultaneously. Likewise, resetting the "system state" for backtracking will reset them simultaneously.

Program 4.10: UML statechart interpreters: analysis variables

```
shared set $\Delta$ ;
shared queue $Q_{evt}$ ;
shared function $vars$ arity 1 ;
shared attribute $\_evtType, \_evtPara,$
                 $\_mode$ ;
derived function $active$ arity 1 = $\{\mathsf{attr}(s, \text{"}name\text{"}) \mapsto (s \in \Delta) \mid s \in \mathsf{S}\}$ ;
derived set $enabledCmplTrans = \{t \in \mathsf{T} \mid \mathsf{attr}(t, \text{"}trigger\text{"}) =$ undef
                 $\wedge (\mathsf{src}_t \in \Delta \cap \mathsf{S}_s) \wedge \mathsf{eval}(\mathsf{attr}(t, \text{"}guard\text{"}), active \cup vars)\}$ ;
derived set $\_nextSteps = \{\surd \mid enabledCmplTrans \neq \emptyset \vee Q_{evt} \neq \emptyset\}$ ;
monitored function $\_step2take$ arity 1 ;
```

In addition to these "state" functions, other analysis variables are also declared in Program 4.10. Among them, the definition of $\_nextSteps$ is somewhat complex as it relies on two auxiliary functions $active$ and $enabledCmplTrans$. "$active$" is a unary function derived from Δ and mapping each state name to a boolean value indicating if this state is active. "*enabledCmplTrans*" is a derived set from Δ and $vars$, consisting of currently enabled completion transitions. A completion transition is enabled if its source is an active simple state and its guard evaluates to true[4]. The guard evaluation is done by parameterizing eval with $active$ and $vars$. According to [159], the existence of enabled completion transitions represents pending completion events. We then define *_nextSteps* to be a singleton set containing a special symbol $\surd$, if there exists an enabled completion transition or a pending event in Q_{evt}. Otherwise, we let *_nextSteps* be an empty set.

4.4.3.3 Initialization

To execute a statechart instance, we need to initialize its state configuration and analysis variables. This process is elaborated by the initialization rule in Program 4.11. This rule is executed once a statechart instance is created.

Program 4.11: UML statechart interpreters: initialization

```
initialize :
        set $\varphi = \{\mathsf{top}\}$ ;
        loop
            do forall $s \in \varphi$ :
                $\Delta := \Delta + \{s\}, \varphi := \varphi - \{s\}$,
                if $s \in \mathsf{S}_{cc}$ then $\varphi := \varphi + \mathsf{subs}(s)$ end,
                if $s \in \mathsf{S}_{sc}$ then $\varphi := \varphi + \{\mathsf{default}(s)\}$ end
            end
        end,
        $vars := \mathsf{attr}(\star, \text{"}VAR\text{"})$,
        $\_mode := \_WAIT$
```

Firstly, a local auxiliary set "φ" is declared for recording the states to be entered next. Initially, it contains only the top state. Then the interpreter runs in a loop until a fixpoint

[4]Note we only consider completion transitions emanating from simple states in this thesis. Strategies to handle complicated cases may be found in [119].

is reached, *i.e.* when no further updates can be made. More specifically in this case, the loop ends when φ is empty. In each run of the loop, all states currently in φ are handled in parallel. Each state is removed from φ and added to Δ (line 5), while one or all of its direct substates are added to φ (lines 6–7). For a sequential composite state, only its unique default state is added, while for a concurrent composite state, all its direct substates are added. In addition, in parallel to the loop execution, $vars$ is assigned to contain the initial values of the declared variables and $\_mode$ is updated to the waiting mode (lines 10–11).

4.4.3.4 Rule "step"

After the initialization, a statechart instance is ready to execute, namely, receive, dispatch and process events. These tasks are specified by an iteratively executed rule "step".

Preliminary definitions Before elaborating the "step" rule, we need to define some additional static functions in Program 4.12. Firstly, given a transition $t \in \mathsf{T}$, "$\mathsf{lca}(t)$" maps it to the *least common ancestor* (LCA) state of t, the lowest composite state that contains all the source and target states of t. The function is defined using an auxiliary macro "ca", which determines the set of common ancestors (by containment) for a given set of states. The states returned by $\mathsf{ca}(\{\mathsf{src}_t, \mathsf{dst}_t\})$ form a chain in the state hierarchy. At the bottom of the chain is $\mathsf{lca}(t)$. In other words, $\mathsf{lca}(t)$ is the state covered by any other state in the chain. The *main source* "$\mathsf{ms}(t)$" of a transition t is either $\mathsf{lca}(t)$ if $\mathsf{lca}(t)$ is a concurrent state, or a direct substate of $\mathsf{lca}(t)$ if $\mathsf{lca}(t)$ is a sequential state. The substate is either the source state of t or a composite state that covers the source state. The *main target* "$\mathsf{mt}(t)$" of t has the same definition except that it ought to cover the target state of t in the last case. It is worth noting that the main source (or target) is the source (or target) state of t, if the state is a substate of $\mathsf{lca}(t)$. In addition, the states that, if active, must be exited when t fires are defined by "$\mathsf{exited}(t)$". This set includes the main source of t and all the states covered by it. As an example, consider Figure 4.3. We have $\mathsf{lca}(t3) = R2$, $\mathsf{ms}(t3) = D$, $\mathsf{mt}(t3) = E$ and $\mathsf{exited}(t3) = \{D\}$. Also, $\mathsf{lca}(t5) = \mathsf{top}$, $\mathsf{ms}(t5) = CS$, $\mathsf{mt}(t5) = F$ and $\mathsf{exited}(t5) = \{CS, R1, R2, A, B, C, D, E\}$.

Program 4.12: UML statechart interpreters: preliminary definitions

```
static function ca arity 1 = $\{X \mapsto \bigcap_{q \in X} \mathsf{cntr}^{+}(q) \mid X \subseteq \mathsf{S}\}$ ;
static function lca arity 1
        $= \{t \mapsto l \mid t \in \mathsf{T}, l \in \mathsf{ca}(\{\mathsf{src}_t, \mathsf{dst}_t\}), \forall l' \in \mathsf{ca}(\{\mathsf{src}_t, \mathsf{dst}_t\}), \mathsf{cover}'(l', l)\}$ ;
static function ms arity 1 $= \{t \mapsto \mathsf{lca}(t) \mid t \in \mathsf{T}, \mathsf{lca}(t) \in \mathsf{S}_{cc}\}$
        $\cup \{t \mapsto m \mid t \in \mathsf{T}, m \in \mathsf{subs}(\mathsf{lca}(t)), \mathsf{cover}'(m, \mathsf{src}_t) \wedge \mathsf{lca}(t) \in \mathsf{S}_{sc}\}$ ;
static function mt arity 1 $= \{t \mapsto \mathsf{lca}(t) \mid t \in \mathsf{T}, \mathsf{lca}(t) \in \mathsf{S}_{cc}\}$
        $\cup \{t \mapsto m \mid t \in \mathsf{T}, m \in \mathsf{subs}(\mathsf{lca}(t)), \mathsf{cover}'(m, \mathsf{dst}_t) \wedge \mathsf{lca}(t) \in \mathsf{S}_{sc}\}$ ;
static relation exited arity 2 = $\{(t, s) \mid t \in \mathsf{T}, s \in \mathsf{S}, \mathsf{cover}'(\mathsf{ms}(t), s)\}$ ;
static relation conflict arity 2 = $\{(t, t') \mid t, t' \in \mathsf{T}, t \neq t'$
        $\wedge\, \mathsf{attr}(t, \text{“}\textit{trigger}\text{”}) = \mathsf{attr}(t', \text{“}\textit{trigger}\text{”}) \wedge \mathsf{exited}(t) \cap \mathsf{exited}(t') \neq \emptyset\}$ ;
static relation priority arity 2 = $\{(t, t') \in \mathsf{conflict} \mid \mathsf{cover}(\mathsf{src}_t, \mathsf{src}_{t'})\}$ ;
```

Two transitions t and t' are *in conflict* if they can be enabled by the same event and also if firing them results in some common states to be exited [159]. Otherwise, they are called *consistent*. “conflict” consists of pairs of transitions that are in conflict with each other. Clearly, conflict is irreflexive and symmetric. For instance in Figure 4.3, we have $(t3, t4), (t4, t3) \in \mathsf{conflict}$.

Relation “priority” specifies the firing priority between two conflicting transitions, where priority is given to the inner-most one [159]. More specifically, a transition originating from a state $s \in \mathsf{S}$ has a higher priority than another transition originating from a state $s' \in \mathsf{S}_c$ such that $\mathsf{cover}(s', s)$. Given two transitions t, t', $(t, t') \in \mathsf{priority}$ indicates that, if both enabled, t' has a higher priority to be executed than t. For example, we have $(t4, t3) \in \mathsf{priority}$ in Figure 4.3.

In the following, we shall use $\mathsf{conflict}(t)$ and $\mathsf{priority}(t)$ for a given transition $t \in \mathsf{T}$ to denote the set of transitions in conflict with t and the set of transitions with priority over t, respectively. We let $\mathsf{conflict}(t) = \emptyset$ (or $\mathsf{priority}(t) = \emptyset$) for $t \in \mathsf{T}$ such that $\nexists (t, t') \in \mathsf{conflict}$ (or $\nexists (t, t') \in \mathsf{priority}$).

Specifying input and run-to-completion steps With the above definitions, we are now able to specify the event receiving, dispatching and processing procedure using Program 4.13.

Program 4.13: UML statechart interpreters: rule "step"

```
rule $step$ :
    if $\_mode = \_INPUT$ then
        import $e$ = function $\{\text{"type"} \mapsto \_evtType, \text{"data"} \mapsto \_evtPara\}$ :
                $Q_{evt} := Q_{evt} + \{e\}$
        end,
        $\_mode := \_WAIT$
    end,
    if $\_mode = \_FIRE \wedge \_step2take = \surd$ then
        if $enabledCmplTrans \neq \emptyset$ then
            $firing(enabledCmplTrans,$ undef$)$
        else
            let $ce = \mathsf{head}(Q_{evt})$,
                $candid = \{t \in \mathsf{T} \mid \mathsf{attr}(t, \text{"trigger"}) = ce(\text{"type"}) \wedge \mathsf{src}_t \in \Delta\}$,
                $env = \{(t, \mathsf{attr}(t, \text{"trgpara"})) \mapsto ce(\text{"data"}) \mid t \in candid\}$,
                $enabled = \{t \in candid \mid \mathsf{eval}(\mathsf{attr}(t, \text{"guard"}), active \cup vars \cup env(t))\}$
            :
                $Q_{evt} := \mathsf{tail}(Q_{evt})$,
                if $enabled \neq \emptyset$ then $firing(enabled, ce)$ end
            end
        end,
        $\_mode := \_WAIT$
    end
```

When an input is available for reception, $\_mode$ should have been set to $\_INPUT$ and $\_evtType$ and $\_evtPara$ to the type and parameter of the input event, respectively. In receiving this input, the interpreter constructs a new function e, mapping "*type*" and "*data*" to the event type and parameter, respectively, and adds it to the event queue (lines 3–5). After that, the interpreter transfers the control back to the analysis tool by setting $\_mode$ to $\_WAIT$.

On the other hand, when $\_mode$ is set to $\_FIRE$ and $\_step2take$ is set to $\surd$ by an analysis tool, it is the time for the interpreter to execute a run-to-completion step (line 8–20). Here, $\surd$ is a special symbol that we have previously used to represent a pending event or completion event in defining $\_nextSteps$ in Program 4.10. At this time, because completion events have priority over normal events to be dispatched [159], this rule first checks for pending completion events. If such events exist, it calls a macro "firing" parameterized by the set of enabled completion transitions and an undefined event representing a completion event. (This macro will be described later.) If no pending completion events exist, this rule dispatches the first event in the queue Q_{evt} (denoted by "head(Q_{evt})") as the current event ce, and computes the set of enabled transitions by ce. Here, two auxiliary functions are used to assist in the computation. The first is a set of possibly enabled transitions $candid$, in which every transition should satisfy two conditions: the trigger matches the event type and the source state is active. The second is a function env mapping each candidate transition and the name of the virtual parameter defined in the trigger into the actual parameter provided by ce. For example in Figure 4.5, suppose $ce = \{\text{“}type\text{”} \mapsto \text{“}d\text{”}, \text{“}data\text{”} \mapsto 1\}$, then $candid$ and env will be $\{t1\}$ and $\{(t1, \text{“}n\text{”}) \mapsto 1\}$, respectively. Next in line 15, the enabled transitions $enabled$ are defined as transitions in $candid$ with satisfied guards. Here, the guard evaluation is conducted by providing eval with the environmental variables defined by unary functions $active$, $vars$ and $env(t)$. After that, ce is removed from the queue Q_{evt} and a macro "firing" parameterized by $enabled$ and ce is used to fire the enabled transitions, if any.

The main job of "firing" is to compute and fire a maximal subset of the enabled transitions which are not in conflict with each other. This subset is called a *maximal consistent set*, namely, the following conditions hold:

- All transitions in the set are enabled and consistent with each other;
- No transition that is enabled and consistent with all transitions in the set is excluded;
- No enabled transition with a higher priority over a transition in the set is excluded.

Program 4.14: UML statechart interpreters: macro “firing”

```
1 $firing(enabled, ce) \equiv$ let $firable = \{t \in enabled \mid \mathsf{priority}(t) \cap enabled = \emptyset\}$ ;
2                         do forall $t \in firable$ ;
3                               $exit(\mathsf{ms}(t))$ ;
4                               $execute(t, ce)$ ;
5                               $enter(\mathsf{mt}(t), \mathsf{dst}_t)$
6                         end
7                    end
```

As we have assumed that all conflicts can be solved using the lower-first priority, the computation of this subset is simplified. It is safe to choose, from the enabled, transitions over which no other enabled have priority. This subset is defined by “$firable$” in Program 4.14 (line 1), where, as stated previously, $\mathsf{priority}(t)$ is a set containing all transitions with priority over a transition t.

With the maximal consistent set computed, we can execute the transitions in the set in parallel due to their mutual consistency. The execution of a single transition consists of a sequence of steps (lines 3–5): exiting the main source, executing the transition effect, and entering the main target.

Exiting the main source also involves exiting all the active substates covered by the main source [159]. The exiting tasks must be conducted in an inside-out order, i.e. the deeper states in the state hierarchy are exited earlier. Program 4.15 shows the exiting procedure for a given main source ms.

First of all, a local set variable φ is declared for recording states to be exited next and initially includes the active leaf states covered by the main source. Next, a fixpoint loop is entered. At each pass of the loop, all states in φ are exited independently. The exiting of a state will result in its removal from Δ. Also, at each pass φ is refreshed with the containers of the exited states (lines 5–6), provided that the states are not the main source. Finally, the loop terminates after the main source has been exited. Special attention should be given to concurrent composite states in that a concurrent state should not be exited before any of its regions. The “with” condition in line 4 is designed to

Program 4.15: UML statechart interpreters: macro "exit"

```
exit(ms) ≡ **begin**
        **set** $\varphi = \Delta \cap \mathsf{S}_s \cap \mathsf{cover}'(ms)$ **:**
        **loop**
            **do forall** $s \in \varphi$ **with** $\mathsf{subs}(s) \cap \Delta = \emptyset$ **:**
                $\Delta := \Delta - \{s\}$, $\varphi := \varphi - \{s\}$,
                **if** $s \neq ms$ **then** $\varphi := \varphi + \{\mathsf{cntr}(s)\}$ **end**
            **end**
        **end**
    **end**
```

constrain the exiting order between a concurrent state and its regions, *i.e.* a state can be exited only if none of its substates is active.

Executing the transition effect involves executing the effect expression as well as event sending. This process is shown in Program 4.16.

To start with, we use an auxiliary environment function env' to link the actual parameter of the current event ce to the virtual parameter's name defined by t. Also, we let evt represent the output event type specified by t and $para$ be the value defined by attribute "$sndpara$" of t. The value is computed using eval and the environmental variables defined by $active$, $vars$ and env'.

Then the transition execution consists of two parallel tasks. On the one hand, the effect expression of t is executed by providing exec with the same environmental variables as above. This may involve updates to the variables in $vars$. On the other hand, $\_evtType$ and $\_evtPara$ need to be modified so as to notify the analysis tool about the event nature of this transition firing. Firstly, if an output event is specified in t (viz. $evt \in \mathsf{O}$), then $\_evtType$ will record this specified type and $\_evtPara$ will be $para$. Otherwise, $\_evtType$ will be τ and $\_evtPara$ will be undefined, indicating this firing is an event internal to the component. As we allow self-addressed internal messages for UML statecharts, evt may be defined. In this case, an internal event is generated and added to the queue Q_{evt}. As usual, the event is an imported function taking evt and $para$ as values.

Program 4.16: UML statechart interpreters: macro "execute"

```
$execute(t, ce) \equiv$
  let $env' = \{\mathsf{attr}(t, \text{"trgpara"}) \mapsto ce(\text{"data"}) \mid ce \neq$ undef$\}$,
      $evt = \mathsf{attr}(t, \text{"sndevt"})$,
      $para = \mathsf{eval}(\mathsf{attr}(t, \text{"sndpara"}), active \cup vars \cup env')$
  :
       $\mathsf{exec}(\mathsf{attr}(t, \text{"effect"}), active \cup vars \cup env')$,
       if $evt \in \mathsf{O}$ then
           $\_evtType := evt, \_evtPara := para$
       else
           $\_evtType := \tau, \_evtPara :=$ undef,
           if $evt \neq$ undef then
               import $e =$ function $\{\text{"type"} \mapsto evt, \text{"data"} \mapsto para\}$ :
                      $Q_{evt} := Q_{evt} + \{e\}$
               end
           end
       end
  end
```

Entering the main target results in states, including the main target and some states covered by it, to be entered in an outside-in order. The entering procedure is similar to the entering of the top state in the initialization rule. The main difference is that this procedure is arranged so that the target state of the executed transition will eventually be entered. Detail about this is given in Program 4.17.

Given a main target mt and a target state ts, the entering starts with mt. As usual, φ is an auxiliary set for pending states to be entered next. It initially contains mt. In each pass of the fix-point loop given next, states in φ are entered in parallel. From lines 8–12, one can see that this differs from the initialization rule in selecting which substate to be entered for a sequential composite state. While the initialization rule selects the default substate, this entering procedure selects the direct substate that either is or covers ts. The default substate is chosen only when no such candidate exists.

Program 4.17: UML statechart interpreters: macro "enter"

```
$enter(mt, ts) \equiv$ begin
            set $\varphi := \{mt\}$ ;
            loop
                do forall $s \in \varphi$ :
                    $\Delta := \Delta + \{s\}$, $\varphi := \varphi - \{s\}$,
                    if $s \in \mathsf{S}_{cc}$ then $\varphi := \varphi + \mathsf{subs}(s)$ end,
                    if $s \in \mathsf{S}_{sc}$ then
                        choose $q \in \mathsf{subs}(s)$ with $\mathsf{cover}'(q, ts)$ :
                            $\varphi := \varphi + \{q\}$
                        else
                            $\varphi := \varphi + \{\mathsf{default}(s)\}$
                        end
                    end
                end
            end
        end
```

4.5 Summary and Related work

In this chapter, we have presented a semantic interpretation approach to heterogeneous systems. This approach specifies a semantic interpreter for each modelling language. The main functions of such an interpreter specification include

- providing the structural characterisation of the current state and transition of a component,
- specifying the computation of the possible transitions (or steps) that a component can make at its current state,
- defining both the process of handling an input event and the process of firing a specified enabled step.

The interpreter for a given language, when parameterized by a component model written in the language, will render sufficient semantic information about the component to support its exhaustive analysis. In other words, given a heterogeneous system, an analysis tool is able to construct the state space of the system, by collaborating with all the component interpreters instantiated for the system. This collaboration will be clarified in the next chapter. As a consequence, on the basis of the semantic framework of discrete-event components developed in Chapter 3, not only can an unambiguous semantics of heterogeneous systems be obtained, but also formal verification is made possible. In this chapter, we have illustrated this semantic interpretation approach with two visual languages: Petri nets and UML statecharts.

As noted in Chapter 2, there are a number of semantics interpretation approaches in the literature, such as [110, 166, 167, 59, 189, 156]. Among them, the work [110] is closest to ours. As stated earlier, our work builds on it and extends it to support formal verification.

In [166, 167], Pezzé and Young focussed on a class of state-transition models and presented an interpretation approach to heterogeneous systems, which facilitates constructing multi-formalism state space analysis tools for these systems. This approach employs a simple common structure (similar to labelled Petri nets) to represent syntactic and static semantic information of heterogeneous models and uses rules to specify their dynamic semantics in terms of transition enabledness (enabling rules), (inter)dependence (matching rules) and execution (firing rules). Our approach is conceptually similar to this approach. However, our approach differs from this approach in that our approach uses ASMs as the interpreter specification language and is able to deal with languages with complex semantics, as demonstrated by UML statecharts.

In [59, 189], Dillon and Stirewalt proposed an approach to the automatic generation of lightweight-analysis engines, called *step analysers*. These generated analysers are able to operate directly on the internal representations of heterogeneous components, and compute all possible next steps for them, using language-specific axioms and inference rules. The set of axioms and inference rules associated with a modelling language constitute the *structural operational semantics* [169] of the language. As a result, analysis tools can utilise these engines to create labelled transitions systems for heterogeneous components and consequently support the formal verification of heterogeneous systems. Step analysers in their approach play a similar role to language interpreters in our approach. However, our approach differs from theirs in the way that the semantics of modelling languages are specified.

In [156], Niu *et al.* introduced an operational semantics template for state-transition specification languages, characterising their common behaviour. Also, they specified 13 kinds of parameter functions for distinct behaviours of languages and allow this template to be parameterized by them so as to specify the step semantics of non-concurrent state machines described in the languages. Further, based on this semantic template, the authors have defined the semantics of 7 composition operators. As a result, an operational semantics of heterogeneous systems is obtained.

5 Implementing Analysis Tools

Based on the semantic interpretation approach presented in Chapter 4, we describe our implementation of analysis tools for component-based heterogeneous systems in this chapter. This implementation is built on the Moses tool suite [65]. We illustrate with two analysis tools, including a simple state space analysis tool and a tool implementing the compositional verification approach proposed in Chapter 3.

In the following, we first give an overview of the Moses tool suite in Section 5.1. We then describe the implementation of a simple and monolithic approach to the verification of heterogeneous systems in Section 5.2. Based on that, we elaborate the implementation issues of our compositional verification approach in Section 5.3. These include both the incorporation of the interface automata formalism in Moses and the tool development for checking the conformance of heterogeneous components, the consistency of interface automata networks, the conformance and safety properties of heterogeneous systems.

5.1 Overview of the Moses Tool Suite

The Moses tool suite, or "Moses" for short, is a set of Java-based software tools developed in the Moses project [2] for supporting the modelling and simulation of heterogeneous systems. It provides the underlying implementation framework for our work. More specifically, this tool suite:

- supports the syntactic and semantic definition of graph-like visual notations. This includes the descriptions of their concrete syntax, well-formedness and formal semantics;
- provides an editing environment customizable by syntactic definitions and consequently supporting the modelling of component-based heterogeneous systems.
- provides a simulation platform customizable by semantic definitions and consequently supporting stepwise execution, animation and state inspection of heterogeneous systems.

The facilities for defining visual notations are typically used by the meta-user to incorporate new visual languages into Moses, while the editor and the simulator are often used to help the user to build system designs using available languages and to validate the resultant designs. The underlying structure of the Moses tool suite is shown as a flow diagram in Figure 5.1, where ovals represent the key components of Moses, rectangles with folded left-bottom corner represent user-supplied documents, quadrangles represent intermediate products, and edges represent the information flow.

Basically, the meta-user has to define the syntax and semantics of a visual notation before any user can actually use it for modelling. The semantics of a notation is defined in

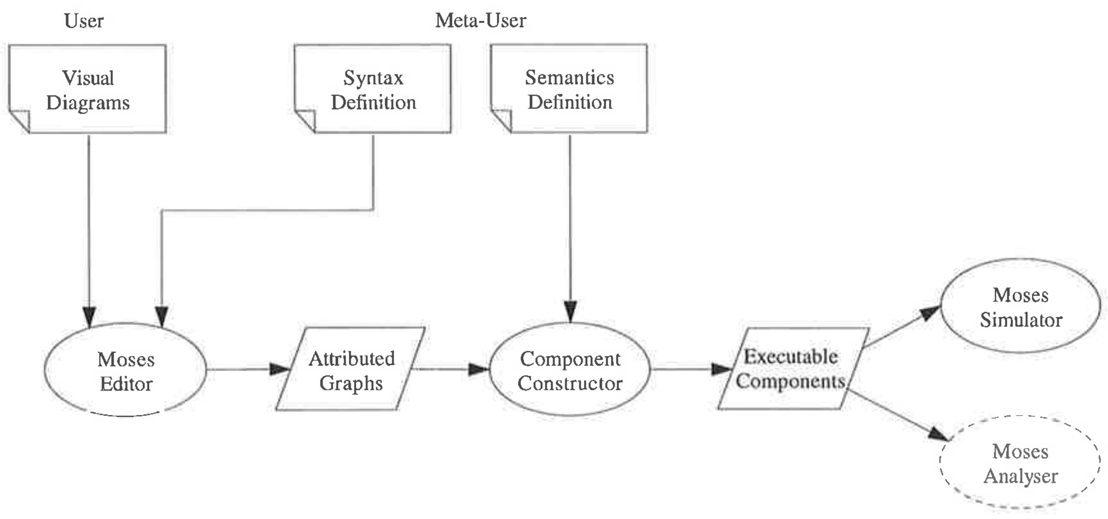


Figure 5.1: Structure of the Moses tool suite

Moses using Object Mapping Automata, as shown previously, while the syntax is defined using a built-in language in Moses, which we shall introduce in Section 5.1.1.

Using a defined notation, the user can build a model (or diagram) of this notation in the Moses editor. This model, if syntactically correct, will be represented as an attributed graph. Based on the attributed graph, the Moses component constructor can build an executable component, which is basically an interpreter of the model instantiated from the predefined OMA semantic specification. This component can then be used for simulation and analysis by the Moses simulator and analyser.

In Section 5.1.2 and 5.1.3, we shall describe the Moses editor and simulator in more detail, respectively. In addition, note that the analyser is shown as a dotted oval in Figure 5.1 to highlight the fact that it is a new addition to Moses. This analyser is the subject of Section 5.2 and 5.3.

5.1.1 Syntactic Definition

For the syntactic definition of various graph-like notations, the Moses tool suite provides a language called the *Graph Type Definition Language* (GTDL) [109]. Basically, given a particular graph-like notation, the GTDL specification defines the kind of attributed graphs used in Moses (cf. Chapter 4). This includes:

- a list of diagram (or graph) attributes, including their names and data types, *e.g.* the attributes "*InputEvent*", "*OutputEvent*" and "*VAR*" of a UML statechart;
- a list of valid kinds of vertices and edges, together with their attributes. The attributes of a vertex or edge include not only its graphical appearance such as shape and size but also its *semantic* attributes, *e.g.* the initial tokens for a Petri net place and the trigger for a UML statechart transition;
- a list of predicates declaring the syntactic constraints (or well-formedness) on instances of this notation.

As a detailed exposition of GTDL is beyond the scope of this thesis, we instead present the Petri net variant described in Section 4.3 as an example to give a flavour of this language (cf. [109] for a complete description of GTDL). The GTDL specification for this Petri net variant is shown in Program 5.1.

Program 5.1: Syntactic specification of Petri nets

```
graph type PetriNet {
        vertex type Place (multiset InitialTokens)
                graphics (string Shape = "Oval",
                          color Color = "black", color FillColor = "white",
                          integer ExtentX = 24, integer ExtentY = 24).

        vertex type Transition (boolexpr guard, expression function)
                graphics (string Shape = "Rectangle",
                          color Color = "black", color FillColor = "white",
                          integer ExtentX = 24, integer ExtentY = 24).

        vertex type InputPort (datatype domain)
                graphics (string Shape = "InputTriangle",
                          color Color = "black", color FillColor = "grey",
                          integer ExtentX = 24, integer ExtentY = 24).

        vertex type OutputPort (datatype range)
                graphics (string Shape = "OutputTriangle",
                          color Color = "black", color FillColor = "grey",
                          integer ExtentX = 24, integer ExtentY = 24).

        edge type Arc (string label)
              graphics (string Head = "ClosedTriangle").

        predicate "Arcs do not emanate from output ports or end at input ports."
                forall a ∈ Arc : src(a) ∉ OutputPort ∧ dst(a) ∉ InputPort end
        predicate "Arcs from input ports must end at places."
                forall a ∈ Arc : src(a) ∈ InputPort ⇒ dst(a) ∈ Place end
        predicate "Arcs from places must end at transitions."
                forall a ∈ Arc : src(a) ∈ Place ⇒ dst(a) ∈ Transition end
        predicate "Arcs from transitions must end at places or output ports."
                forall a ∈ Arc : src(a) ∈ Transition ⇒
                                 dst(a) ∈ Place ∪ OutputPort end
}
```

In lines 2–23, four kinds of vertices and one kind of edges are declared: places, transitions, input ports, output ports and arcs. First of all, their graphical appearance is described in the "graphics" section. For example, it is claimed that all places are displayed as white ovals with black border, initially 24 wide and 24 high. Furthermore, immediately following the name of each kind, the semantic attributes are declared. For instance, a place has an attribute "*initialTokens*" specifying the multiset of tokens initially residing in it. A transition has a boolean expression as the guard and an expression as the function.

After these declarations, four well-formedness predicates are given in lines 25–33, constraining the arcs between vertices. These include a requirement on the absence of arcs emanating from output ports or ending at input ports, and also the constraints on the targets of arcs emanating from input ports, places and transitions.

5.1.2 Diagram Editing

The Moses tool suite provides a generic diagram editor for building models. Above all, this editor does not depend on any particular notation (or graph type) but manipulates diagrams at an abstract level, in particular, as attributed graphs (as in Definition 44). It provides the basic functions for diagram manipulation such as the insertion, deletion and layout of vertices and edges.

Furthermore, the editor can be parameterized by a GTDL specification and then behave in a way specific to the notation. For example in Program 5.1, each vertex type has a "Shape" attribute. This identifies a piece of code predefined in Moses for drawing the picture representing a vertex of this type. Also, the colors and initial size of the picture can be specified by other attributes such as "Color", "FillColor", "ExtentX" and "ExtentY".

In addition, the editor provides facilities for the user to modify the attributes associated with vertices, edges and even a diagram itself, especially the semantic attributes. Examples include the modification of the guard expression or color for a Petri net transition and the modification of variable declarations for a UML statechart.

Apart from the editing facilities, the editor is also able to check the well-formedness of the diagram being edited, using the given GTDL predicates. A violation of the well-formedness will be recorded and shown to the user for correction.

5.1.3 Diagram Simulation

After building a well-formed model, the user often wants to validate its dynamic behaviour in order to uncover potential design defects and gain confidence in the design. For this purpose, Moses provides a simulator for the stepwise execution, animation and state inspection of the model.

Basically, the simulator uses the Moses component constructor to construct an interpreter for this model, from the predefined semantic specification of its notation. Then the simulator executes the interpreter step by step, firing one of the enabled steps reported by the interpreter at a time. It also provides an animator visualizing the states and state changes of this model and thus the whole execution process.

The simulation of a component-based system is more complex than a single component. This involves not only generating interpreters for all components models but also coordinating the concurrent execution of components and relaying the exchanged messages between components. For technical reasons, Moses has adopted an interleaving semantics. That is, only one enabled step can be scheduled for firing at a time and any transition firing is atomic. Basically, all the enabled steps of components are considered equally. One of them is chosen for firing randomly. Also, messages exchanged between communicating components are only transferred immediately after transition firings.

5.2 Approach to Verification

The existing implementation of the Moses tool suite does not include a verification (or model checking) tool. Although a simulator is included, it can only provide partial validation of system models, due to the lack of support for system state storage and exhaustive state space exploration. Also, it provides no facilities for property specification and automatic verification.

In order to support formal verification, Chapter 4 has extended the way that interpreters are specified. Furthermore, a state space exploration tool is designed, which takes full advantage of the additional facilities of interpreters to support exhaustive enumeration of system states. In addition, a side-effect free language is defined to specify system properties against which systems can be verified. More specifically, an existing language in Moses, called the ***Expression LANguage*** (ELAN) [110], is reused for property specification. In the following sections, these issues will be addressed in more detail.

5.2.1 State Space Exploration

In Section 4.2, we have designed a contract between component interpreters and analysis tools. This means that the state space exploration tool has to abide by the contract. For the sake of simplicity, we have chosen to implement a simple exploration algorithm which employs no state space reduction techniques.

Consider a complete component-based system (or semantically, a closed DEC network) $D = (P, \gamma)$. Let μ_p be the instantiated component interpreter of $p \in P$. Then a DFS exploration algorithm for D is described by pseudo-code in Program 5.2. Its ultimate aim is to construct the synchronised product of D, consisting of an initial state s^0, a set S of states reachable from s^0 and a set Δ of steps between these states.

As noted previously, a (global) state of the system (including s^0) is a vector made up of (local) states of the components, we build a global state as the product of all the local states in this algorithm. Further, we use a stack *pending* to record the unexplored steps, and a set *compSteps* to record each pair of component and enabled step. In particular, every pair $\langle p, e\rangle$ in *compSteps* satisfies the condition that step e is enabled in component p at its current local state.

At the beginning, this algorithm waits until the initialization of all component interpreters have terminated, as required in Figure 4.1. Then it initializes s^0, S and Δ. In particular, it assigns to s^0 the current global state of the system, namely, a vector of the current local states of components. The algorithm then uses a macro definition "*checkProperty*" to check some user-specified properties at the initial state. We defer a description of this macro until the property specification language ELAN has been introduced in Section 5.2.2.

Program 5.2: A simple state space exploration algorithm for a closed system D

```
wait until $\forall p \in P, \mu_p.\_mode = \_WAIT$
$s^0 \leftarrow \Pi_{p \in P}\, \mu_p.\_state,\ S \leftarrow \{s^0\},\ \Delta \leftarrow \emptyset$
$checkProperty$

$compSteps \leftarrow \{\langle p, e\rangle \mid p \in P, e \in \mu_p.\_nextSteps\}$
$pending \leftarrow \{\langle s^0, compSteps\rangle\}$
while $pending \neq \emptyset$ do
    $\langle \mathrm{s}, untaken\rangle \leftarrow pending.pop()$
    if $untaken \neq \emptyset$ then
        $\langle dec, stp\rangle \leftarrow untaken.removeAny()$
        $pending.push(\langle \mathrm{s}, untaken\rangle);$
        foreach $p \in P$ do $\mu_p.\_state \leftarrow \pi_p(\mathrm{s})$ end
        $\mu_{dec}.\_step2take \leftarrow stp$
        $\mu_{dec}.\_mode \leftarrow \_FIRE$
        wait until $\mu_{dec}.\_mode = \_WAIT$

        if $\mu_{dec}.\_evtType = \tau$ then $event \leftarrow \tau$
        else
            foreach $p \in P$ do
                if $\exists \mathrm{f} \in \gamma, \pi_{dec}(\mathrm{f}) = \mu_{dec}.\_evtType \wedge p \in \eta_{\mathrm{f}}$ then
                    $\mu_p.\_evtType \leftarrow \pi_p(\mathrm{f})$
                    $\mu_p.\_evtPara \leftarrow \mu_{dec}.\_evtPara$
                    $\mu_p.\_mode \leftarrow \_INPUT$
                end
            end
            wait until $\forall p \in P, \mu_p.\_mode = \_WAIT$
            $event \leftarrow \langle \mu_{dec}.\_evtType, \mu_{dec}.\_evtPara\rangle$
        end

        $\mathrm{s}' \leftarrow \Pi_{p \in P}\, \mu_p.\_state$
        $\Delta \leftarrow \Delta + \{\langle \mathrm{s}, event, \mathrm{s}'\rangle\}$
        if $\mathrm{s}' \notin S$ then
            $S \leftarrow S + \{\mathrm{s}'\}$
            $compSteps \leftarrow \{\langle p, e\rangle \mid p \in P, e \in \mu_p.\_nextSteps\}$
            $pending.push(\langle \mathrm{s}', compSteps\rangle);$
            $checkProperty$
        end
    end
end
```

After property checking, the set *compSteps* is computed (line 5). This basically adds a pair $\langle p, e\rangle$ into *compSteps* for every component $p \in P$ and step $e \in \mu_p.\_nextSteps$. As stated previously, variable $\mu_p.\_nextSteps$ is a set consisting of all the steps enabled at the current state of p. For a Petri net component, *_nextSteps* consists of all the currently enabled transitions, while for a UML statechart component, *_nextSteps* is either an empty set or a singleton set consisting of the special symbol $\surd$.

Next in line 6, the *pending* stack is initialized to contain a pair composed of s^0 and *compSteps*. Now this algorithm is ready to conduct a full state space exploration in a while loop (lines 7–39). The exploration terminates only when *pending* is empty. At each pass of this loop, a single enabled step *stp* of a component *dec* is taken (if any). More specifically, suppose the top element of *pending* is $\langle \mathrm{s}, untaken\rangle$, then $\langle dec, stp\rangle$ must be an element of *untaken*. Executing a step *stp* involves three steps. Firstly, this algorithm sets each component back to its local state corresponding to s, viz. the original state where *stp* was enabled (line 12). Secondly, it asks *dec* to fire *stp* and waits for the completion (lines 13–15). Finally, it transfers the output data from *dec* to all the connected components (lines 19–27), assuming an output is produced (*i.e.* $\mu_{dec}.\_evtType \neq \tau$). The data transfer to the interpreter μ_p of a component p involves updating its variables *_evtType*, *_evtPara* and *_mode* (to *_INPUT*) as well as waiting for the completion of the input reception by μ_p.

After *stp* is taken, this algorithm needs to update Δ, S and *pending* accordingly. It first uses an auxiliary vector s′ to store the current global state and then adds into Δ a step leaving s and targeting s′ (lines 30–31). This step is labelled by *event*, the event of *dec* that just occurred (cf. lines 17, 27). In addition, if s′ has not yet been visited, this algorithm adds it to S and pushes a pair $\langle \mathrm{s}', compSteps\rangle$ into *pending* for the next iteration (lines 33–35), where *compSteps* is the set of currently enabled steps at s′. Likewise, properties can be checked here using macro *checkProperty*.

Note that although designed for non-hierarchical systems, Program 5.2 is also applicable to hierarchical systems since they all have equivalent non-hierarchical counterparts (cf. Section 3.3.1). Note also that this algorithm makes no assumption about the nature of individual modelling languages. This opens the possibility of other model checking techniques developed for homogeneous systems to be applied to heterogeneous systems. However, this is beyond the scope of this thesis and thus left for future work.

5.2.2 Specifying Safety Properties

As already mentioned, we use an embedded language in Moses, ELAN, for property specification. ELAN [110] is a functional language with a simple, side-effect free semantics. It supports a number of basic and structured data types such as booleans, integers, real numbers, strings, sets, lists and maps. It also supports the common numerical, logical and set operators.

To give an impression on how to specify a system property, we consider the component-based system in Figure 3.3 (page 46). Suppose the adder and the doubler are modelled by Figure 4.2 (page 87) and 4.5 (page 99), respectively. To require the doubler to respect the input assumption of the adder, we can write the following constraint on the markings of the adder's places:

$$\#pa <= 1 \wedge \#pb <= 1,$$

where pa denotes the multiset of tokens currently residing in place pa and $\#pa$ is the size of this multiset. This restricts the number of tokens in each of places pa and pb to at most one at all times. Furthermore, to require the user to respect the input assumption of the doubler, we can write for the doubler:

$$\neg\, ERR,$$

where ERR is a boolean variable indicating whether the UML state "ERR" is active. This requires that the user never send a request again before getting the result from the doubler. Additionally, to relate the adder and the doubler, we can write

$$doubler.S2 \implies (\#adder.pa = 1 \wedge \#adder.pb = 0), \tag{5.1}$$

where the name of a variable in a component is prefixed by the component name. This means that whenever the doubler is at $S2$ there must be a token in place pa but no token in place pb of the adder.

As claimed in Chapter 3, in this thesis we only consider state-based properties which involve only the present behaviour but no past or future behaviour. Hence we do not introduce temporal operators such as "X", "F" and "U" as in CTL [36] and LTL [170]. In other words, the safety properties we consider can be evaluated merely on the basis of a single state of a system. These properties, in the terminology of temporal logics, are either atomic propositions or their negation, conjunction or disjunction. Therefore, a property φ written in ELAN can be considered equivalent to a LTL expression "G φ" or a CTL expression "AG φ".

5.2.3 Verifying Safety Properties

To enable the verification of safety properties, we extend interpreter specifications with a derived *environment* function called "$\_env$", which maps the name of each (local) state variable in a component into its current value. The basic idea is to use a component's own interpreter to interpret its local variables for property evaluation. In particular, we define $\_env$ for Petri net interpreters as follows:

$$\underline{\textbf{derived}}\ \underline{\textbf{function}}\ \_env = \{\mathsf{attr}(p, \text{"}name\text{"}) \mapsto \mathcal{M}(p) \mid p \in \mathsf{P}\}$$
$$\cup \{\mathsf{attr}(t, \text{"}name\text{"}) \mapsto (t \in \_nextSteps) \mid t \in \mathsf{T}\},$$

where each place is mapped to its marking and each transition is mapped to a boolean indicating if it is enabled. Similarly, for UML statechart interpreters, we add:

$$\underline{\textbf{derived}}\ \underline{\textbf{function}}\ \_env = active \cup vars,$$

where each state is mapped to a boolean indicating whether it is active, and each diagram variable is mapped to its value.

Next, we build an array referencing the map *_env* for every component. As a consequence, when checking properties, we can interpret state variables for different components using their respective environment functions. For instance, when checking property 5.1, we shall use the local variable environment of the doubler to evaluate $doubler.S2$ and that of the adder to evaluate $adder.pa$ and $adder.pb$.

Program 5.3: Macro “checkProperty(φ)”

```
1 $env \leftarrow \emptyset$
2 foreach $cn.vn \in VarNames$ do
3         $env \leftarrow env + \{cn.vn \mapsto \mathsf{eval}(vn, CompMap(cn).\_env)\}$
4 if $\neg\,\mathsf{eval}(\varphi, env)$ then
5         /* report the property violation. */
```

Program 5.3 shows the process of checking a property φ. It assumes $CompMap$ is a map from each component name to the component interpreter (assuming each component is uniquely named), $VarNames$ is a list of variable names appearing in φ and each variable name is in the form of “$CompName.VariableName$” as shown in formula 5.1. To begin with, this algorithm builds a new environment env, mapping every variable name present in φ into its value (lines 1–3). This involves the evaluation of each variable name using eval and the local environment of its corresponding component interpreter. As we have assumed in Chapter 4, eval is a platform function for expression evaluation. Next in line 4, this algorithm evaluates property φ using this newly built environment env. If φ does not hold, it will report the violation.

5.3 Approach to Compositional Verification

In the previous sections, we have described the process of defining graph-like modelling languages in Moses and our implementation strategies for a simple state space exploration and checking tool. Based on the developed facilities, in this section, we shall discuss the implementation issues of the compositional verification method proposed in Chapter 3.

As noted, the key in this compositional method is the introduction of interface automata for specifying the interface protocol of components. Thus the first thing in our implementation is to incorporate the visual notation of IAs into Moses. This uses the same syntactic and semantic definition methods presented in Section 5.1 and Chapter 4, respectively, and will be detailed in Section 5.3.1.

In addition, as stated in Chapter 3, to determine the basic properties of a component-based system, the conformance of heterogeneous components with their respective IAs as well as the consistency of the derived IA network has to be ensured. Therefore, after the incorporation of IAs, algorithms for conformance and consistency checking will be presented in Section 5.3.2–5.3.4.

Furthermore, algorithms implementing the relevant theoretical work from Chapter 3 for proving safety properties of component-based systems will be described in Section 5.3.5.

5.3.1 Incorporating the IA Formalism

As described in Section 3.2.2, IAs are a graph-like formalism with a very simple semantics. Using the method presented in Section 5.1.1, we can easily define the syntax and well-formedness of the IA formalism in Moses. The resultant GTDL specification is shown in Program 5.4.

As shown, two graph attributes are associated with an IA model for declaring its observable events (line 2). After that, initial states, states and transitions in addition to well-formed rules are declared. In particular, three predicates are enforced to ensure the disjointness of input and output events, the uniqueness of initial state, and the validity of transition labels in an IA model, respectively (lines 13–18).

Next, applying the methods described in Section 4.3.3 and 4.4.3, we can easily specify a semantic interpreter for IAs. As we actually use the input-universal RTSs and most abstract implementations of IAs in our compositional verification approach, we define semantic interpreters in terms of these two derivatives rather than the IAs themselves. Given the attributed graph $\mathsf{G} = (\mathsf{V}, \mathsf{E}, \mathsf{src}, \mathsf{dst}, \mathsf{attr})$ for a well-formed IA, the interpreter for

Program 5.4: Syntactic specification of IAs

```
graph type InterfaceAutomata {
        attribute set InputEvent, OutputEvent.

        vertex type InitialState ()
                graphics (string Shape = "ArrowDot", color Color = "black",
                            integer ExtentX = 8, integer ExtentY = 8).
        vertex type State ()
                graphics (string Shape = "Dot", color Color = "black",
                            integer ExtentX = 8, integer ExtentY = 8).
        edge type Transition (string label)
              graphics (string Head = "ClosedTriangle").

        predicate "Input and output events are disjoint."
                      InputEvent ∩ OutputEvent = ∅
        predicate "A unique initial state must exist."
                      #InitialState = 1
        predicate "A transition label must be either an input or output event."
             forall t ∈ Transition : attr(t, "label") ∈ InputEvent ∪ OutputEvent end
}
```

its input-universal RTS semantics is specified by Program 5.5, where $\perp$ represents the error state.

This specification consists of three parts. The first part (lines 1–7) defines all the analysis variables required in Section 4.2. In particular, *_nextSteps* includes all the output transitions (or edges) enabled at the current state *_state*. The first part also defines a local variable environment *_env* imposed in Section 5.2.3 to enable the property checking for IAs. From lines 6–7, one can see that *_env* conditionally contains two string elements: "*error*" if the current state is $\perp$ and "*waiting*" if a currently enabled input transition exists.

The second part (lines 9–13) specifies the initialization rule, which sets *_state* to be the unique initial state defined in the IA and *_mode* to be the waiting mode.

Program 5.5: Specification of the input-universal RTS semantics of IAs

```
**shared attribute** $\_state$ **:**
**shared attribute** $\_evtType, \_evtPara,$
                     $\_mode$ **:**
**derived set** $\_nextSteps = \{t \in \mathsf{E} \mid \mathsf{src}_t = \_state \wedge \mathsf{attr}(t, \text{“label”}) \in OutputEvent\}$ **:**
**monitored attribute** $\_step2take$ **:**
**derived set** $\_env = \{\text{“error”} \mid \_state = \bot\}$
                     $\cup \{\text{“waiting”} \mid \exists t \in \mathsf{E}, \mathsf{src}_t = \_state \wedge \mathsf{attr}(t, \text{“label”}) \in InputEvent\}$ **:**

**initialize :**
        **choose** $s \in \mathsf{V}$ **with** $\mathsf{attr}(s, \text{“type”}) = \text{“InitialState”}$ **:**
                $\_state := s$
        **end**,
        $\_mode := \_WAIT$

**rule** $step$ **:**
    **if** $\_state \neq \bot$ **then**
        **if** $\_mode = \_INPUT$ **then**
            **choose** $t \in \mathsf{E}$ **with** $\mathsf{attr}(\mathsf{src}_t, \text{“label”}) = \_evtType$ **:**
                    $\_state := \mathsf{dst}_t$
            **else**
                    $\_state := \bot$
            **end**
        **end**,
        **if** $\_mode = \_FIRE$ **then**
            **let** $t = \_step2take$ **:**
                $\_state := \mathsf{dst}_t,$
                $\_evtType := \mathsf{attr}(t, \text{“label”}),\ \_evtPara :=$ **undef**
            **end**
        **end**
    **end**,
    $\_mode := \_WAIT$
```

The last part (lines 15–31) specifies the "step" rule, elaborating the process of input reception and output generation for the IA. More specifically, when receiving an input event specified at the current state, the IA will move to the target state of a transition whose label matches the event (line 19). Such a transition is unique due to the determinism of IAs. On the other hand, if receiving an unspecified input event, the error state will be entered as in Definition 6. In addition, if scheduled to fire a transition t, the IA will enter the target state of t and set the last event $\langle \_evtType, \_evtPara \rangle$ with the label of t and an undefined parameter. Note that if the IA is already in the error state, nothing will happen except for executing a common task, *i.e.* updating $\_mode$ to the waiting mode.

Furthermore, if given a typing function θ mapping each input/output event to a set of valid values, we can define an interpreter implementing the MAI semantics for the IA, viz. its most abstract implementation with respect to θ. This requires a modification of Program 5.5 on the definition of $\_nextSteps$ (line 4) and the output generation (lines 25–28). The details are given in Program 5.6 and 5.7, respectively. In Program 5.6, $\_nextSteps$ is redefined to contain transition-value pairs, each of which includes an output transition t enabled at the current state $\_state$ and a valid output value v from set $\theta(e)$ where e is the labelling event of t. Correspondingly, if scheduled to fire a step $\langle t, v \rangle$, the MAI will execute Program 5.7, outputing data value v together with event e. This differs from the output generation described in Program 5.5 where no data value is involved.

Program 5.6: Specification of the MAI semantics of IAs: $\_nextSteps$ definition

```
4 derived set $\_nextSteps = \{\langle t, v \rangle \mid t \in \mathsf{E}, v \in \theta(\mathsf{attr}(t, \text{“}label\text{”})),$
5                                    $\mathsf{src}_t = \_state \wedge \mathsf{attr}(t, \text{“}label\text{”}) \in OutputEvent\}$ ;
```

5.3.2 Conformance Checking for Components

With the syntax and semantics of IAs defined in Moses, we are now able to employ IAs for specifying the interface protocols of components. To employ our compositional method for the verification of component-based systems, we first need to ensure the conformance and live conformance of individual components with their associated IAs.

Program 5.7: Specification of the MAI semantics of IAs: Output generation

```
        let $\langle t, v\rangle = \_step2take$ :
            $\_state := \mathsf{dst}_t$,
            $\_evtType := \mathsf{attr}(t, \text{“}label\text{”})$, $\_evtPara := v$
        end,
```

As noted in Section 3.2.4, these two kinds of conformance can be determined merely with the local state space of each component. Also, the local state space of a component C with respect to an IA A is the synchronised product of a closed network of two components: C and the MAI of the mirror of A. As the construction of the mirror from A is a syntactical transformation, we assume M is the constructed mirror. We then instantiate an interpreter of M using the MAI semantics (with respect to a typing function θ as in Definition 20) to construct the local state space of C.

To ensure the conformance of C with A, we know from Theorem 1 that it is sufficient to check the absence of the error state in the local state space of C. This task is equivalent to checking a property:

$$\neg\, \mu_M.error$$

in the network composed of C and the MAI of M, assuming μ_M is the instantiated interpreter of M. Clearly, this check can be executed using the state space exploration and checking algorithms presented in Program 5.2 and 5.3.

To ensure the live conformance of C with A, we need to check an additional requirement that C is able to produce at least one of the enabled input events of M. Suppose we have ensured the conformance of C with A using the above-mentioned method. Let $S_\otimes$ and $\Delta_\otimes$ be the sets of states and steps in the constructed local state space of C, respectively. Then we determine the live conformance of C with A using a backward search, detailed by Program 5.8. Note that for simplicity we assume $\Sigma_A^I \subseteq \alpha_C^I$ and $\Sigma_A^O \supseteq \alpha_C^O$ in this program.

Firstly, the states violating the first condition of Theorem 2 are listed as *suspect* states. That is to say, at such states, M is waiting for some input event. This is determined by evaluating the “*waiting*” variable at every state $s \in S_\otimes$ using the corresponding local

Program 5.8: Checking the live conformance of a component C

```
suspect ← $\{s \in S_\otimes \mid \mathsf{eval}(\text{"waiting"}, getEnv(\mu_M, \pi_M(s)))\}$
exits ← $\{s \in suspect \mid \exists e \in \Sigma_C^O, (s, e, s') \in \Delta_\otimes\}$

pending ← exits
safe ← $\emptyset$
while $pending \neq \emptyset$ do
        $safe \leftarrow safe + pending$
        $pending \leftarrow \{s \in S_\otimes \mid \exists s' \in pending, e \in \Sigma_C^H, (s, e, s') \in \Delta_\otimes\}$
end

if $safe = suspect$ then
      /* live conformance holds. */
end
```

variable environment of μ_M. This local environment is obtained by calling a function "*getEnv*" defined in Program 5.9. As shown, this function call will first reset the current state of μ_M to $\pi_M(s)$ and then return the local variable environment of μ_M at $\pi_M(s)$.

Program 5.9: Function "getEnv"

```
function $getEnv(\mu, s)$
            $\mu.\_state \leftarrow s$
            return $\mu.\_env$
end
```

Next in Program 5.8, some suspect states, at which C is able to produce an output, are distinguished, since they clearly satisfy the precondition of Theorem 2 and thus are safe for proving the live conformance. A set *exits* is used to hold these states.

In addition to *suspect* and *exits*, there are two auxiliary sets *pending* and $safe$ in Program 5.8. They are used to record all the calculated "safe" states so far. More specifically, *pending* records the newly calculated, while $safe$ holds the previously calculated. Initially, *pending* is equal to *exits* and $safe$ is empty.

After the initialization, a while loop will be executed by Program 5.8. At each pass of this loop, all the states in *pending* are moved into $safe$ and *pending* is reloaded with their immediate predecessors reachable via an internal event of C for the next iteration. Clearly, these predecessors are also safe for proving the live conformance, satisfying the precondition of Theorem 2. Finally, this while loop terminates when *pending* is empty, *i.e.* no more "safe" states were found.

At that time, if all the suspect states have been proved to be "safe", we can conclude that C live-conforms to A. Otherwise, the live conformance does not hold.

5.3.3 Consistency Checking for IA Networks

As noted in Section 3.3.3 and 3.3.4, to verify the basic properties of a component-based system, we need to ensure not only the conformance of the components with their respective IAs but also the consistency of the derived IA network. This section is dedicated to the implementation issues for constructing the state spaces (or synchronised products) of IA networks and checking their consistency.

To construct the state space of an IA network $N = (W, R)$, we instantiate interpreters for the constituent IAs using their input-universal RTS semantics and employ the state space exploration algorithm in Program 5.2.

Furthermore, to determine the consistency of an IA network, we invoke Program 5.2 with the following property to check:

$$\forall an \in IANames, \neg\, an.error,$$

where we assume $IANames$ is a set consisting of the names of all the constituent IAs.

In addition, to enable the live consistency checking of N, we also need to test the condition stated in Definition 29, *i.e.* the absence of deadlock states, states where the network is blocked but some IA is still waiting for an input. Suppose the consistency of N has been proved as above. Let μ_a be the instantiated interpreter of $a \in W$. Then Program 5.10 shows the algorithm to determine its live consistency.

Program 5.10: Checking the live consistency of an IA network N

```
terminals ← $\{s \in S_N \mid \nexists \langle s, e, s' \rangle \in \Delta_N\}$
deadlocks ← $\{s \in terminals \mid \exists a \in W, \text{eval}(\text{"}waiting\text{"}, getEnv(\mu_a, \pi_a(s)))\}$
if deadlocks $= \emptyset$ then
    /* live consistency hold. */
end
```

This algorithm uses *terminals* to store all terminal states in the network and *deadlocks* to store the terminal states that violate the condition of Definition 29. The violation by a terminal state s is interpreted as the existence of an IA a whose "*waiting*" variable is true. As usual, the "*waiting*" variable of an IA a is evaluated using the local environment of μ_a at its local state $\pi_a(s)$. This algorithm claims the live consistency of the IA network if *deadlocks* contains no elements in the end.

5.3.4 Conformance Checking for Open Systems

As demonstrated by Theorem 7, the algorithms presented previously are sufficient to determine the conformance of open component-based systems (or semantically open DEC networks). However, these algorithms are not enough to determine their live conformance. In the following, we present a solution to this problem.

Consider an open component-based system D and an IA A such that the conformance of D with A have been ensured compositionally as shown previously. Let N be the derived IA network of D, M be the mirror of A, E_v and E_i be as in Theorem 8. Then Program 5.11 shows the process of compositionally verifying the live conformance of D with A.

Similar to Program 5.8, this algorithm consists of three parts and uses a backward search starting from "exit" states. The first part (lines 1–2) marks "suspect" and "exit" states in S_N, while the second part (lines 4–9) identifies "safe" states by executing a while

Program 5.11: Checking the live conformance of an open system D

```
1  $suspect \leftarrow \{\mathrm{s} \in S_N \mid \mathsf{eval}(\text{“}waiting\text{”}, getEnv(\mu_M, \pi_M(\mathrm{s})))\}$
2  $exits \leftarrow \{\mathrm{s} \in suspect \mid (\exists e \in E_v, (\mathrm{s}, e, \mathrm{s}') \in \Delta_N) \wedge (\nexists e' \in E_i, (\mathrm{s}, e', \mathrm{s}') \in \Delta_N)\}$
3
4  $pending \leftarrow exits$
5  $safe \leftarrow \emptyset$
6  while $pending \neq \emptyset \wedge pending \cap safe = \emptyset$ do
7          $safe \leftarrow safe + pending$
8          $pending \leftarrow \{\mathrm{s} \in S_N \mid \exists \mathrm{s}' \in pending, e \in E_i, (\mathrm{s}, e, \mathrm{s}') \in \Delta_N\}$
9  end
10
11 if $safe = suspect$ then
12         /* live conformance holds. */
13 end
```

loop. The last part reports the satisfaction of the live conformance if all suspect states prove to be safe.

Unlike Program 5.8, this algorithm needs to check more conditions. More specifically, to be safe, a suspect state of N must satisfy both Condition 1 and 2 of Theorem 8. That is, it must not be part of an internal loop, and it must be able to reach an exit state via an internal trace. This is reflected by both the computation of exit states (line 2) and the adaptation of the while condition (line 6). In particular, exit states are the suspect states with only observable outgoing steps. They are thus safe for proving the live conformance, according to the above requirements. Also, apart from the empty test on *pending*, the while condition also includes an empty test on “$pending \cap safe$” to detect internal loops. This means that there must be an internal loop if and only if a safe state is visited again during the backward search.

Finally, this algorithm is able to conclude the live conformance of D if all suspect states prove to be safe. Otherwise, no conclusive answer is given, due to the abstraction of IAs from components.

5.3.5 Verifying System-Wide Safety Properties

To prove a safety property local to a component, we can simply call macro "*checkProperty*" as defined in Program 5.3 while constructing the local state space of the component. However, to prove a system-wide safety property, we often cannot do so due to the potential state space explosion. Fortunately, we can employ the compositional verification approach proposed in Section 3.3.5 for DEC networks, the semantic model for component-based heterogeneous systems.

In this section, we present our implementation strategy for this approach, assuming that the consistency of these systems (or networks) has been ensured using the algorithms proposed earlier. That is to say, the local state spaces of the components and the state space of the derived IA networks have been constructed.

Consider a consistent network $D = (\alpha, \theta, P, \gamma)$ with respect to β and a safety property clause φ on D as in Definition 42, that is, $\varphi = \varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_k$ $(k \leq |P|)$ such that all φ_j $(1 \leq j \leq k)$ are local properties of distinct components. Let N be the derived IA network of D (with respect to β), S_N be the state space of N, $1 \leq j \leq k$, μ_j be the interpreter of the DEC $p_j \in P$ to which φ_j is local, $S_{j\otimes}$ be the local state space of p_j, $a_j = \beta(p_j)$ and S_{a_j} be the state space of a_j. Then the algorithm for the compositional verification of φ is shown in Program 5.12.

Program 5.12: Verifying a safety property clause φ

```
foreach $j : 1 \leq j \leq k$ do
        $\overline{sat}_j = \{s_j \in S_{a_j} \mid \exists \langle q_j, s_j \rangle \in S_{j\otimes}, \neg \mathsf{eval}(\varphi_j, getEnv(\mu_j, q_j))\}$
        $\mathtt{sat}_j \leftarrow S_{a_j} \setminus \overline{sat}_j$
end

$safe \leftarrow \{\mathrm{s} \in S_N \mid (\exists a \in W, \pi_a(\mathrm{s}) = \bot) \vee (\exists j : 1 \leq j \leq k, \pi_j(\mathrm{s}) \in \mathtt{sat}_j)\}$
if $safe = S_N$ then
    /* $\varphi$ holds. */
end
```

This algorithm consists of three steps. The first step (lines 1–4) computes, for every $j\colon 1 \leq j \leq k$, a set $\mathtt{sat}_j$ of the IA states in S_{a_j} that satisfy φ_j in $S_{j\otimes}$. This computation uses an exclusive method to exclude from S_{a_j} the IA states that do not satisfy φ_j, since these states are computationally cheaper to compute than those satisfying φ_j. Here, the non-satisfaction of φ_j by a state $\langle q_j, s_j \rangle \in S_{j\otimes}$ is evaluated using the local environment of μ_j at q_j returned by function $getEnv$.

Next, the second step (line 6) calculates a set of "safe" states in S_N, where the precondition of Theorem 11 and 12 is satisfied. Clearly, this requires only a single scan on S_N.

Finally, if all states in S_N prove to be "safe", this algorithm reports the satisfaction of φ. Otherwise, no conclusive answer is given due to the abstraction introduced by IAs, as stated earlier.

5.4 Summary and Discussion

In this chapter, based on the interpreters developed in Chapter 4 for heterogeneous components, we have described the implementation of a simple and monolithic verification approach to component-based heterogeneous systems. This includes an algorithm for constructing the system state space (Program 5.2) and solutions to the specification and verification of safety properties.

In addition, we have presented algorithms implementing the compositional verification approach proposed in Chapter 3. As noted previously, given a component-based heterogeneous system, the basic properties can be proved by independently employing Program 5.2 and 5.8 to ensure the (live) conformance of each component, and Program 5.2 and 5.10 to ensure the (live) consistency of its derived IA network. Further, the conformance of an open system with an IA can be ensured using Program 5.11. As a result, independent analysis at different hierarchical levels of the system is supported. In addition, while the local properties of primitive components can be proved relatively easily within their local state spaces using Program 5.2, system-wide properties of both closed and open systems can be proved compositionally using their derived IA networks by Program 5.12.

At the time of writing, we have implemented the algorithms presented above using Java in the context of the Moses tool suite. The Moses tool suite is available at [3]. Our analysis tools are available in a side branch of the repository and will soon be merged into the main branch.

It should be noted that the implementation of our compositional approach builds on a simple state space explorer described in Program 5.2. This explorer currently does not employ any advanced state space reduction technique. However, we believe, the employment of such techniques in its future version would allow us to further reduce the size of state spaces that need to be handled by our compositional approach.

6 Case Study: the Production Cell

In the previous chapters, we have presented a foundation for the specification and verification of heterogeneous systems and a compositional approach to verifying such systems, and studied the implementation issues in the context of the Moses tool suite. The ultimate aim is to alleviate the state space explosion problem while model-checking complex systems. In this chapter, we demonstrate the power of the proposed techniques by applying them to the verification of a non-trivial system: the Production Cell [138].

The remainder of this chapter is structured as follows. We first give a brief description of the Production Cell in Section 6.1. We next elaborate our design methodology in Section 6.2, and then apply the compositional verification methods developed earlier to the resultant design in Section 6.3. Finally, we present discussions and the related work in Section 6.4.

6.1 Task Description

The production cell case study, posed in [138], was derived from a metal processing plant. The main task of the cell is to forge metal blanks in a press. The blanks are transported to and removed from the press through the collaboration of five other machines in the cell: a feed belt, an elevating rotary table, a robot with two extendable arms, a deposit belt and a travelling crane. Figure 6.1 shows the top view of the cell (taken from [88]).

The production cycle of a metal blank is as follows. When the feed belt conveys the blank to the table, the table rotates and lifts the blank to a position where the robot can

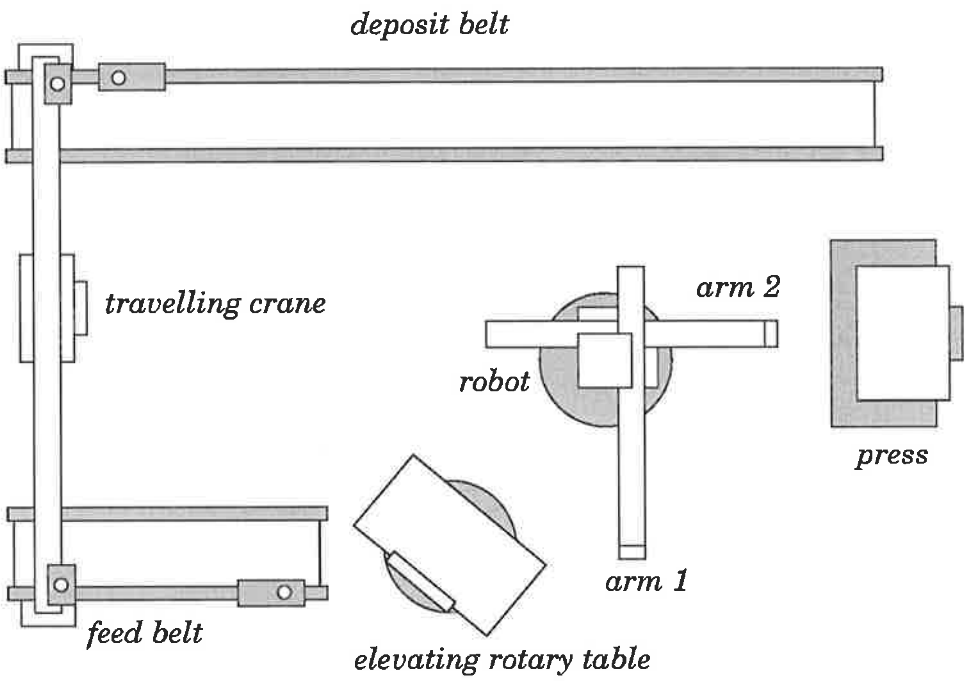


Figure 6.1: Top view of the Production Cell

pick it up using its first arm. After picking up the blank, the robot rotates anticlockwise and places the blank into the open press. After that, the press closes to forge the blank and then opens to its lower position, the same height as the second arm of the robot. Next, the robot picks up the forged blank using the second arm and places it onto the deposit belt. The deposit belt then transports it to the end. The system is made closed and self-contained (with a finite state space) by the addition of a crane which fetches the blank from the end of the deposit belt and returns it to the beginning of the feed belt.

6.1.1 Constituent Machines

To perform the intended task of the cell, each of the six machines is fitted with a number of actuators and sensors. The machine learns the current position of its own and a loaded blank by reading from its sensors, and transports or forges the blank by controlling its actuators. In total, this cell involves 13 actuators and 14 sensors. In the following, we give the detail about each machine as well as its actuators and sensors.

Feed belt is powered by an electric motor. By starting or stopping the motor, the belt can transport and deliver blanks to the table. A photoelectric cell is installed at the end of the belt to detect the arrival and departure of a blank at the final part of the belt;

Elevating rotary table is fitted with two electric motors since both its vertical movement and rotation are necessary. By controlling these two motors, the table is able to rotate and lift up blanks in order to enable the first robot arm to pick them up. In addition, two (boolean valued) switches are used to detect if the table has reached its top and bottom positions. Also, an analog potentiometer is installed to detect the rotation angle of the table;

Robot comprises two extendable arms mounted orthogonally at different levels. Each arm is fitted with an electric motor which allows it to extend and retract horizontally. It is also fitted with an electromagnet at the end for picking up blanks. Furthermore, to transport blanks between neighboring machines, a swivelling motor is installed in the robot, allowing the robot to rotate between four angles: where the first arm points to the table, where the second arm points to the press, where the second arm points to the deposit belt, and where the first arm points to the press. In addition, three analog potentiometers are installed to detect the current extensions of the arms and the current angle of the robot;

Press consists of two horizontal plates, with the lower plate being movable along the vertical axis. It forges blanks by pressing its lower plate against the upper plate. Apart from the upper position, the lower plate has two other positions: a middle position for loading by the first robot arm and a lower position for unloading by the second robot arm. Therefore, in the press, an electric motor is fitted to take charge of the vertical movement of the lower plate. Three switches are used to indicate the current position of the lower plate;

Deposit belt is similar to the feed belt. It is powered by an electric motor which allows the belt to transport blanks to the end in order for the crane to pick them up. Also, a photoelectric cell is installed near the end of the belt to detect the entry of a blank into the final part of the belt; the cell is one blank away from the end of the belt so that a blank will still be on the belt after having passed the cell. This makes the deposit belt different from the feed belt.

Travelling crane is used to transport blanks from the deposit belt back to the feed belt. It has an electromagnet gripper powered by two electric motors and movable in both the horizontal and vertical directions. The horizontal movement is to make up the distance between the two belts, while the vertical movement is to make up their difference in height. Further, the crane is fitted with two switches, indicating if the gripper is above any of the belts, and a potentiometer, measuring the current height of the gripper.

6.1.2 Requirements

According to [138], the Production Cell case study involves a variety of requirements. Among them, those concerning the safety of the cell are the most important since their violation may result in damage to machines or injury to people. In this thesis, we concentrate on the safety requirements.

Basically, the safety requirements of the cell are consequences of restrictions on machine movement, avoidance of machine collisions, prevention of blanks being dropped outside safe areas, and insurance of sufficient distance between blanks [138]. Examples include:

- The robot must not be rotated clockwise if its first arm points towards the table, nor be rotated anticlockwise if its first arm points towards the press;
- The press may only close when no robot arm is positioned inside it;
- The feed belt may only convey a blank through its light barrier if the table is stopped and in the loading position.
- A new blank may only be put on the deposit belt, if the former blank has arrived at the end of the deposit belt.

In the remainder of this chapter, we shall use these requirements as an example to demonstrate the key issues in the case study. Meanwhile, for the sake of brevity, we delegate an extensive exposition and discussion of other safety requirements to Appendix A.

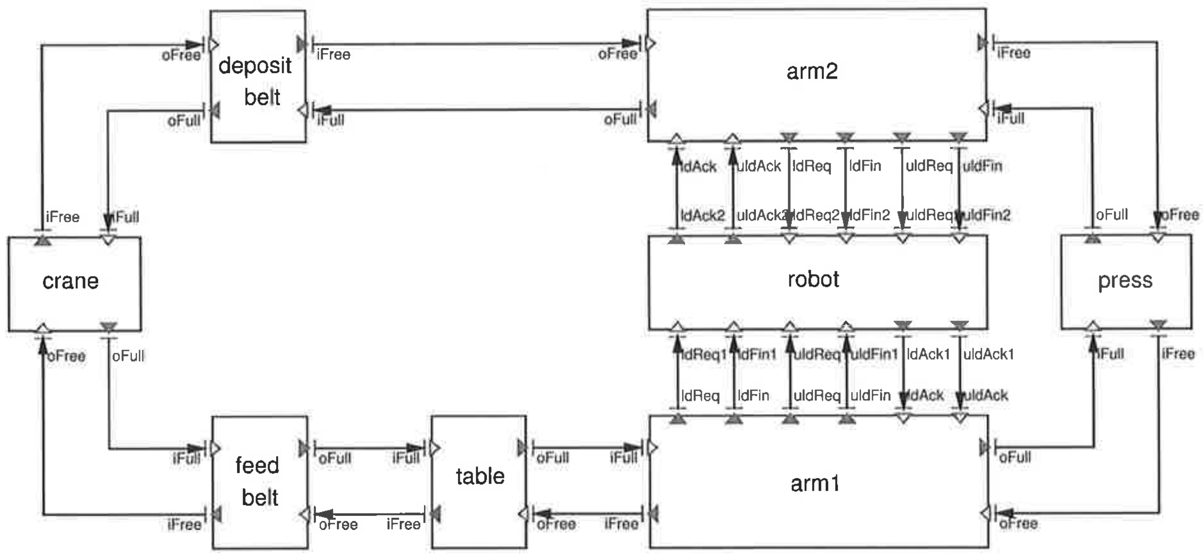


Figure 6.2: Architecture of the Production Cell

6.2 Component-Based Design

Our design of the Production Cell is based on the Petri net model proposed by Heiner and Deussen [88, 89]. Figure 6.2 depicts the architecture of the design as a block diagram.

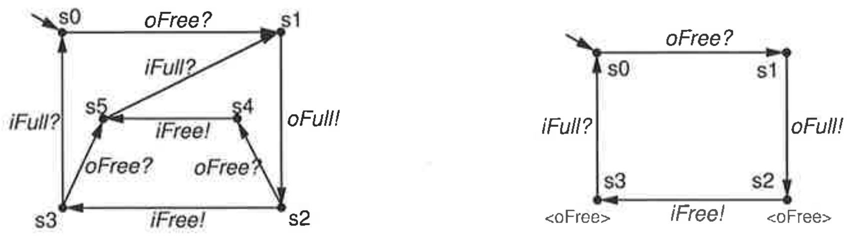
In this design, each machine in the cell corresponds to a component except the robot machine. In [88], the robot machine was designed as two arm components, each of which involves a control logic for swivelling the robot machine to its loading or unloading angle. This is counter-intuitive to a component-based system view. Instead, we build an additional component "*robot*" which is in charge of robot swivelling at the request of the two arms. This makes the arm components identical and thus helps produce reusable implementations. Hence the robot machine now corresponds to three components in our design: "*arm1*", "*arm2*" and "*robot*".

To simplify matters, in this thesis, we only consider a closed system with five blanks. Initially, three blanks reside on the feed belt, the table, and the press, respectively, while the other two reside on the deposit belt, one at each end. However, we believe, other variants of the Production Cell can be verified in the same way.

6.2.1 Protocol Design

To coordinate the concurrent execution of components, we follow Heiner and Deussen's approach [88]. That is, components communicate according to the producer-consumer protocol.

Firstly, consider the feed belt and the deposit belt. Figure 6.3(a) shows the protocol specification IA for both of them. In this figure, "$iFull$" and "$iFree$" are the exchanged events between the belt and its upstream component (*e.g.* the crane in the case of the feed belt). These events are used to unlock and lock the input region of the belt, respectively. In particular, $iFull$ is an input event used to signal the completion of blank delivery by the upstream component, while $iFree$ is an output event used to notify the upstream about the readiness of the belt to accept blanks. Also, the events "$oFree$" and "$oFull$" are exchanged between the belt and its downstream component (*e.g.* the table in the case of the feed belt). These events are used to unlock and lock the output region of the belt. Put differently, $oFree$ is an input event used to signal the readiness of the downstream component to accept blanks, while $oFull$ is an output event used to notify the downstream component of the completion of blank delivery by the belt.



(a) Full version (b) Compact version

Figure 6.3: IAs for the feed/deposit belt

The IA in Figure 6.3(a) specifies the need of the feed/deposit belt for simultaneous control of its input and output regions [88]. In other words, the belts cannot deliver/convey blanks unless the two regions are locked. More specifically, at states $s1$ and $s4$, the belt holds both input and output locks. On the contrary, it holds no locks at $s3$. In addition, it holds only the input lock at $s0$ and $s2$, and only the output lock at $s5$.

Because of the reactive nature of IAs, the interleaving of input and output events sometimes makes IA diagrams messy, even for simple protocols. Take the IA in Figure 6.3(a) as an example. Input event $oFree$ is acceptable at states $s0$, $s2$ and $s3$, but its occurrence is independent of the occurrence of event sequence $iFree \cdot iFull$ from $s2$. Therefore, $oFree$ is interleaved with these two events in the diagram, and two states $s4$ and $s5$ are used to buffer $oFree$ so as to model the acceptance of $oFree$ at states $s2$ and $s3$. However, due to the addition of $s4$ and $s5$, the understanding of the key issues of this protocol is hindered. To improve upon this, we introduce a special attribute for both states $s2$ and $s3$ to specify an input event that requires buffering, *i.e.* $oFree$, and remove $s4$ and $s5$ from the diagram. The ultimate diagram is shown in Figure 6.3(b), where the special attribute is denoted as "$\langle oFree \rangle$". As a result, the graph is neater and easier to understand. Note that we only regard the special attribute as a syntactic shortcut to facilitate further description. All the checking tasks described later are conducted using the full versions of IAs.

Apart from the variant in Figure 6.3, there are two variants of the producer-consumer protocol in the design. They are shown in Figure 6.4(a) and 6.4(c). Figure 6.4(a) states that the table/press must exclusively handle inputs and outputs, in other words, blank reception and delivery. That is, they need to lock their input and output regions in order to change positions. More specifically, the table/press holds both input and output locks at states $s0$ and $s2$, while it holds one input lock and one output lock at states $s1$ and $s3$, respectively. Furthermore, Figure 6.4(c) indicates that the crane needs only an independent control of its input and output regions. That is, it requires only one lock at a time to do its work. In particular, the crane holds no locks at states $s1$ and $s3$, while it holds one input lock and one output lock at states $s0$ and $s2$, respectively,

To communicate with the neighbouring components, arms also follow the protocol shown in Figure 6.4(c). In addition, they coordinate with the robot component under a resource sharing protocol as depicted in Figure 6.4(b) and 6.4(d). The protocol ensures that the sensitive operations, such as blank pickup, blank release and robot swivelling, can be exclusively executed without interruption. Basically, a semaphore is exchanged between

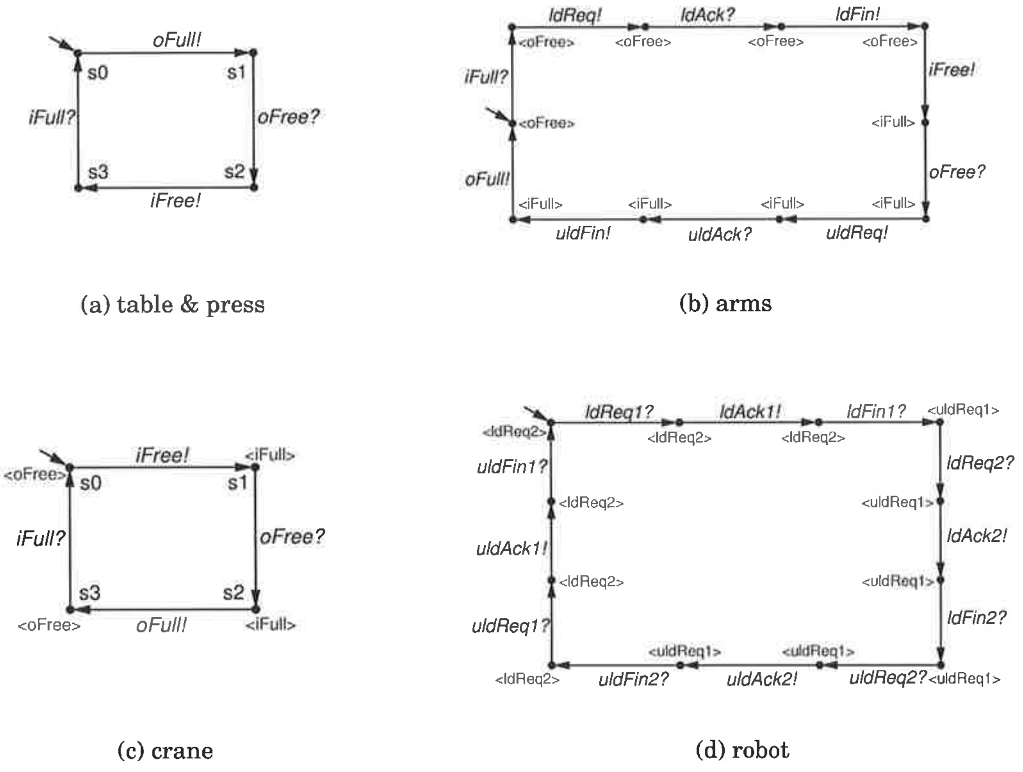


Figure 6.4: IAs for the other components

them so that only one component, the one having the semaphore, can conduct sensitive operations. Initially, *robot* owns the semaphore. An arm can ask for the semaphore by sending a request via "*ldReq*" or "*uldReq*". The request also indicates the angle at which the arm wants the robot to be, e.g. *ldReq* (or *uldReq*) for the loading (or unloading) angle of the arm. Once *robot* has swivelled to the requested angle, it hands over the semaphore to the arm by sending "*ldAck*" or "*uldAck*". When the arm finishes sensitive operations, it returns the semaphore via "*ldFin*" or "*uldFin*". Furthermore, upon receiving a request, *robot* may process it immediately or buffer it for later processing. When both arms request the semaphore at the same time, *robot* will choose which one to serve. The action to take depends on the current angle of the robot and the availability of the semaphore.

6.2.2 Component Design

With the interface automata formally describing the interaction protocols of components, we can now independently design each component with respect to an IA. Black-token Petri nets are used here as the modelling language.

As noted previously, the Production Cell system involves many physical devices such as actuators and sensors. Although different in functionality, size and shape, these devices have similar control procedures, which Heiner and Deussen named the elementary motion control [88, 89].

In the following, we first iterate the essential strategy proposed by Heiner and Deussen for modelling the control procedure of an elementary motion. We then present the models for five key components including the belts, the arms and the robot, revealing their control logic on the actuators and sensors as well as their collaboration with other components. The models for other components, such as the table, the press and the crane, can be found in Appendix A.

6.2.2.1 Elementary Motion Control

In the Production Cell, every machine motion consists of a sequence of elementary motions, each of which involves only one actuator and one sensor [88]. To model each machine, it is essential to understand the control procedure of an elementary motion. This procedure is described by the Petri net in Figure 6.5.

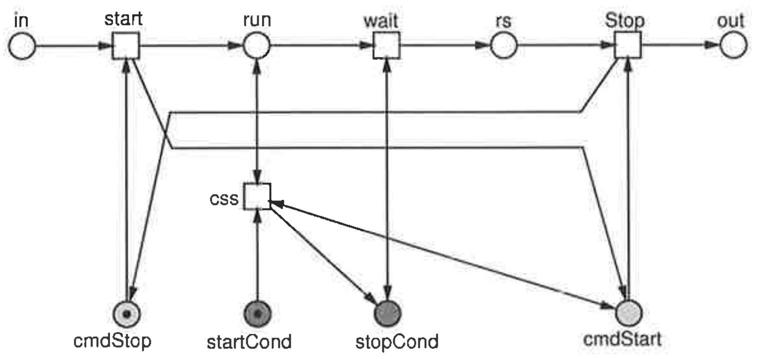


Figure 6.5: Control model for elementary motions

In Figure 6.5, commands for controlling an actuator are modelled by complementary places "*cmdStart*" and "*cmdStop*". The presence of a token in such a place means that the actuator has received the command and is in an associated state. Likewise, the start and stop conditions of an elementary motion are modelled as complementary places "*startCond*" and "*stopCond*", and a token in such a place represents that the reading of the sensor meets the start/stop condition. The start and stop conditions are only modified by a sensor state controller "*css*" when "*cmdStart*" is flagged. Here, "*css*" stands for "change sensor state". Next, the control procedure of an elementary motion is modelled by the link from "*start*", "*run*", "*wait*", "*rs*" to "*stop*", where "*rs*" stands for "ready to stop". Lastly, places "*in*" and "*out*" are used to indicate the initialization and completion of the motion.

Basically, when initialized, the control procedure begins only if the actuator has previously stopped. It empties *cmdStop* and deposits a token into *cmdStart* to start the actuator, and then waits for the stop condition. We assume here that in the physical world the stop condition will eventually become true as a result of the running of the actuator. This is represented by transitions *css*, which will make the stop condition to be true, once the actuator is started and the start condition is satisfied. After that, transitions *wait* and *stop* will be sequentially fired. This will remove the token from *cmdStart* and add a token to *cmdStop* and *out*. Finally, the actuator is stopped and the motion is completed.

6.2.2.2 Feed Belt

As mentioned previously, the job of the feed belt is to transport and deliver blanks onto the table. It is fitted with an electric motor for driving the belt and a sensor for detecting blanks at the final part of the belt. The component model for the belt is shown in Figure 6.6. This model is borrowed from [88] but flattened and enhanced with input/output ports. Further, in order to improve the efficiency of the belt and the table, in contrast to [88], we allow the feed belt to transport a loaded blank without locking its output region. Note that the belt still needs to lock the output region for blank delivery.

In this model, starting and stopping the electric motor is described by depositing a token into complementary places "*cmdStart*" and "*cmdStop*", respectively. The reading of the sensor is modelled by complementary places "*lightBarrierFalse*" and "*lightBarrier-True*", where a token in *lightBarrierTrue* indicates that a blank is at the final part of

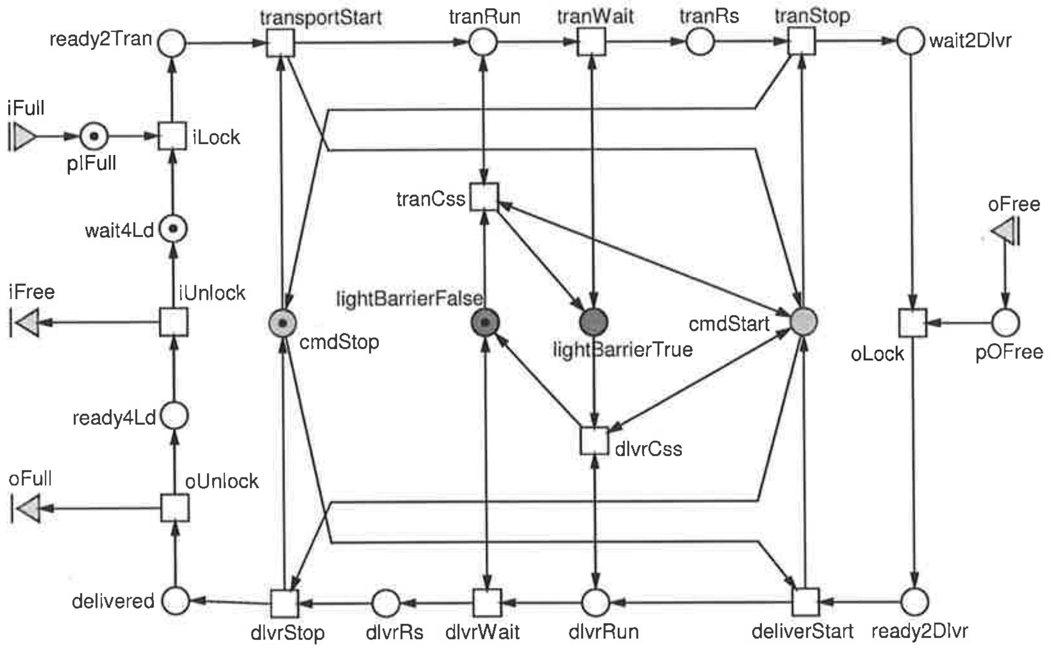


Figure 6.6: Component model for the feed belt

the belt. In addition, transitions "*transportStart*", "*tranWait*" and "*tranStop*" model the control procedure for transporting a blank from the beginning to the end. Likewise, transitions "*dlvrStart*", "*dlvrWait*" and "*dlvrStop*" model the delivery process.

Initially, the belt is idle and stopped with one blank at the beginning. As shown in Figure 6.6, it has a token in *pIFull*, meaning that its upstream (*i.e.* the crane) has unlocked its input region. In other words, the crane has completed the delivery of the last blank. After locking its input region by firing *iLock*, the belt proceeds to transport the loaded blank. The transportation finishes when the light barrier becomes true, meaning that the loaded blank has arrived at the final part of the belt. The belt then waits for an *oFree* event to deliver the loaded blank. As stated, *oFree* indicates that the downstream (*i.e.* the table) has unlocked the output region of the belt and is ready to accept a blank. After receiving *oFree*, the belt will lock its output region and then proceed to deliver the blank. The delivery finishes when the light barrier becomes false, meaning that the loaded blank has left the final part of the belt. After that, the belt will sequentially produce two output events *oFull* and *iFree* to transfer the input and output locks to its neighbouring components by firing *oUnlock* and *iUnlock*, and wait for lock release by its neighbours.

It is worth noting that some invariants are present in this model, *e.g.* the complementary places noted above. These can be ensured with purely local information, while other invariants, *e.g.* the boundedness of places, would require information on how the environment behaves. The principle proposed in Section 3.3.5 assumes the component has an environment that is as helpful as is defined by the relevant IA. This allows us to prove invariants of the latter kind in the local state space of the belt, as will be demonstrated later in Section 6.3.

6.2.2.3 Deposit Belt

With similar functionality and fittings of actuators and sensors, the deposit belt has a component implementation similar to the feed belt's. Figure 6.7 shows the model of the deposit belt.

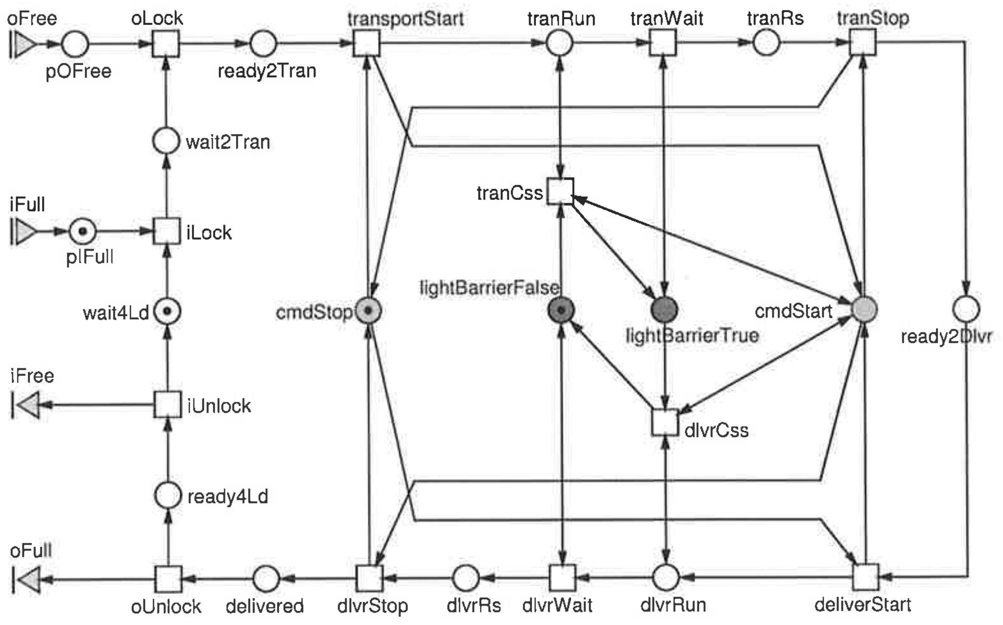


Figure 6.7: Component model for the deposit belt

This model differs from the feed belt's in the location of the transition "*oLock*". More specifically, while the feed belt locks its output region after blank transportation, the deposit belt will lock its output region before that. Here, the blank transportation for both the belts is modelled by the group of transitions: *transportStart*, *tranWait* and *tranStop*. The different location of *oLock* in the deposit belt is due to the fact that, before being picked up by the crane, a delivered blank is still on the belt, *i.e.* at the final part of the belt behind the light barrier. Therefore, the deposit belt has to wait for the crane to pick up the delivered blank, before it can transport a newly loaded blank to the end. In contrast, for the feed belt, a delivered blank is no longer on the belt, but on the table. Hence, the feed belt is free to transport a new blank after the delivery.

6.2.2.4 Arms

In designing the robot machine, we distribute its actuators and sensors into three components: "*arm1*", "*arm2*" and "*robot*". The arm components manage the actuators and sensors participating in arm extension, blank grasp, and arm retraction, while the robot component manages those engaged in robot swivelling. In this section, we present the model for the arm components, while in the next section, we describe the implementation of *robot*.

As noted previously, each arm of the robot machine is fitted with a motor for arm extension and retraction, an analog potentiometer for detecting the current arm extension, and an electromagnet for blank grasp. Although this makes the arms more complicated than the belts, a similar design approach can still be applied. The resultant model of the arms is shown in Figure 6.8.

In this model, the arm extension is simplified into three positions: the retracted, pickup, and release extensions. These are modelled by three complementary places "*atRetractExt*", "*atPickupExt*" and "*atReleaseExt*", respectively. Furthermore, the control commands of the motor for stopping, moving forward and backward are modelled by three complementary places "*cmdStop*", "*cmdGoForward*" and "*cmdGoBackward*", respectively. Also, the control commands of the electromagnet are modelled by two complementary places "*cmdMagOn*" (for blank grasp) and "*cmdMagOff*" (for blank release).

The control procedure of arm extension for picking up a blank consists of three steps. The first controls the motor for extending the arm from the normal retracted position to the pickup position. This is modelled by the link from "*ldExtendStart*" and "*lxWait*"

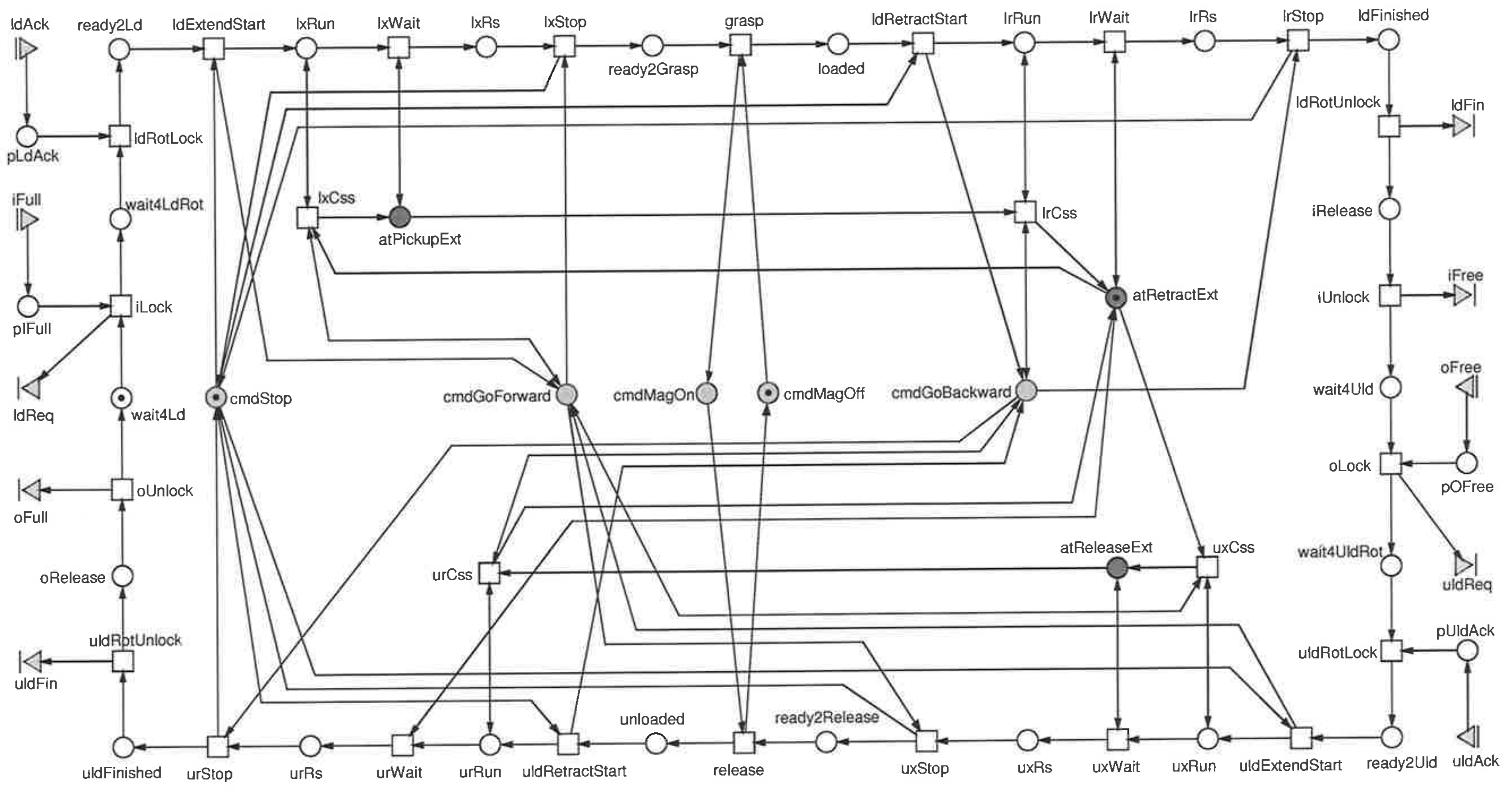


Figure 6.8: Component model for the arms

to "*lxStop*" at the top line of Figure 6.8. The second step controls the electromagnet for blank grasp, modelled by transition "*grasp*". The last step retracts the arm back to the normal extension. This is modelled by a group of transitions including "*ldRetractStart*", "*lrWait*" and "*lrStop*". In addition, the control procedure of arm extension for releasing the loaded blank is similar, as shown at the bottom of Figure 6.8.

Initially, an arm waits for its upstream component to be ready for blank delivery. More specifically, it waits for an incoming message from port "*iFull*". As stated earlier, this message is sent after the upstream has reached its own unloading position. In particular, for *arm*1, this means that the table has lifted up a blank to the top position and is stopped at the unloading angle, while, for *arm*2, this means that the lower plate of the press has reached the lower position.

If receiving a message (or a black token) from *iFull*, transition "*iLock*" will be enabled. Firing *iLock* results in a loading request to be sent to the robot component via "*ldReq*". As will be shown in the next section, upon receiving this request, *robot* will rotate to the angle needed for the arm to load and then respond with a "*ldAck*" message. After receiving this message, the arm will have the rotation semaphore for executing sensitive operations. More specifically, it is ready to load the blank from its upstream. To load a blank, the arm will conduct a sequence of operations for arm extension, blank grasp and arm retraction, by firing the transitions at the top line of Figure 6.8. It will then release the semaphore back to *robot* by a *ldFin* message and release its input lock to its upstream by an *iFree* message.

After that, a similar procedure will be followed by the arm to unload a blank. This involves waiting for the readiness of its downstream component to accept a blank, requesting *robot* to swivel to its unloading angle, executing sensitive operations for arm extension, blank release and arm retraction, and finally releasing the rotation semaphore and its output lock.

6.2.2.5 Robot

As noted previously, the main job of the robot component is to swivel the robot machine under the request of the arms. For the swivelling, four angles are particularly important, including the loading and unloading angles of both arms. The analog potentiometer fitted allows *robot* to detect the current angle of the robot. In designing the component, we abstract the reading of the potentiometer into discrete values representing the above four angles. Furthermore, we model the arrival of the robot at these angles using four complementary places: "*atLdAngle*1", "*atLdAngle*2", "*atUldAngle*2" and "*atUldAngle*1", where 1 stands for *arm*1 and 2 for *arm*2. These places are coloured dark grey in the model of the robot component shown in Figure 6.9.

In addition, the robot component has to follow a resource sharing protocol in order to collaborate with the arms. More specifically, *robot* has to own the rotation semaphore in order to swivel the robot machine. The semaphore is modelled as a token initially residing in place "*rotSafe*" in Figure 6.9 (at the bottom-left corner). It is exchanged between *robot* and the arm components so that the one having the semaphore can execute sensitive operations, *e.g.* "swivelling" in the case of *robot*. The transfer of the semaphore to the arms is modelled by four transitions "*grantLd*1", "*grantLd*2", "*grantUld*2" and "*grantUld*1". Also, the semaphore is returned via ports "*ldFin*1", "*ldFin*2", "*uldFin*2" and "*uldFin*1", respectively, after the arms have completed their tasks.

Also in Figure 6.9, the control commands for robot swivelling are modelled by three complementary places "*cmdStop*", "*cmdRotClockwise*" and "*cmdRotAnticlockwise*". The control procedures of swivelling between the four angles are modelled by four groups of transitions $\{ldRotState1, lrWait1, lrStop1\}$, $\{ldRotState2, lrWait2, lrStop2\}$, $\{uldRotState2, urWait2, urStop2\}$ and $\{uldRotState1, urWait1, urStop1\}$. Take the *ldRotStart*1 group as an example. To start such a procedure, four conditions have to be satisfied:

- *arm*1 has requested for its loading angle but permission has not yet been given, *i.e.* *pLdReq*1 is not empty;
- *robot* currently owns the rotation semaphore, *i.e.* *rotSafe* is not empty;
- *arm*1 has completed the unloading task, *i.e.* *uldFinished*1 is not empty;
- the rotation motor is stopped, *i.e.* *cmdRotStop* is not empty.

Initially, the robot component is stopped at the loading angle of *arm*1 and waits for a request from the arm components. When receiving a loading request from *arm*1, it will transfer the rotation semaphore to *arm*1 by firing *grantLd*1. When the semaphore is returned, *robot* will swivel to the loading angle of *arm*2, if requested, by firing the

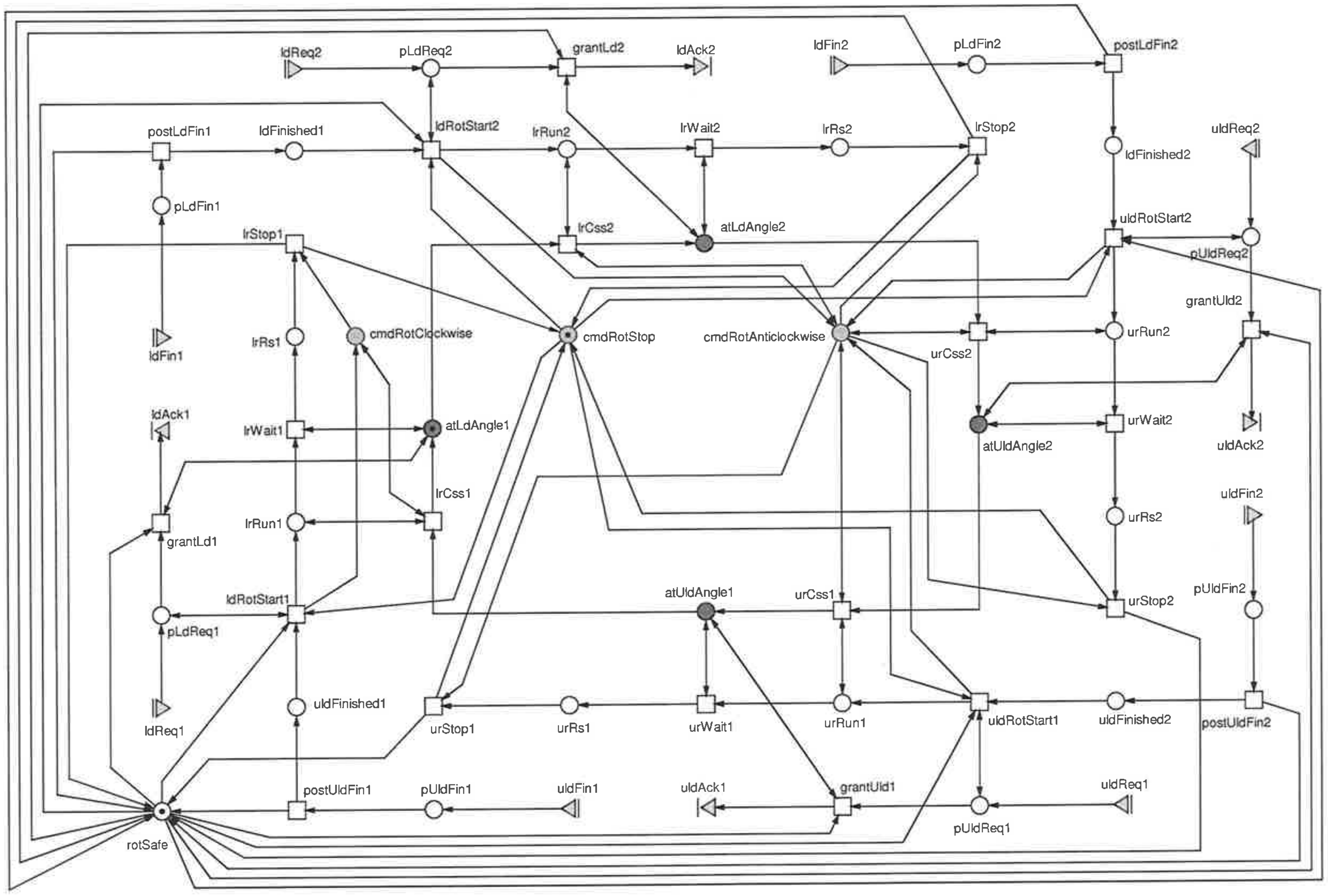


Figure 6.9: Component model for the robot

transitions in the *ldRotStart*2 group. Similarly, it will swivel to the unloading angle of *arm*2 and then swivel to the unloading angle of *arm*1, if requested.

6.3 Compositional Verification

In the last section, we have presented a component-based design approach to the Production Cell. In this section, we present our approach to the verification of the resultant design. The properties under consideration include the basic properties as well as the safety requirements.

To check these properties, one could adopt a monolithic approach and build the whole state space which, as will be shown in Table 6.2, has over 2 million states and 9 million transitions. This would no doubt result in a high computational complexity in verification. To avoid this, we adopt the compositional verification approach proposed in Chapter 3.

6.3.1 Verifying Basic Properties

As defined in Section 3.3.3, the basic properties of a component-based system refer to its consistency and deadlock freedom. To check these properties, we first check each component for conformance and live conformance with its corresponding IA. This ensures that, in a system where the input assumptions of the IA are respected, the component does not break the output guarantees specified by the IA and also that it is free from deadlock. From Section 3.2.4, we know that the (live) conformance of a component can be determined in the local state space of the component. Using our analysis tools implemented in the Moses tool suite, we have constructed the local state space of every component in the Production Cell design, and have successfully proved its conformance and live conformance with its corresponding IA. Table 6.1 shows the size of the resultant component local state spaces. Clearly, it is not computationally expensive to prove the (live) conformance of each component.

Next, we build an abstract system (or network) composed of all the IAs under the same interconnections as shown in Figure 6.2. The IAs include the one in Figure 6.3(a) and the full versions of those in Figure 6.4. We then construct the synchronised product (or state

	States	Transitions
feed belt	21	28
table	22	22
robot	74	119
arm1/arm2	60	90
press	18	18
deposit belt	17	20
crane	80	136

Table 6.1: Local state spaces of the components

	States	Transitions
Component network	2,068,884	9,548,785
IA network	692	1,191

Table 6.2: Global state spaces of the networks

space) of the formed IA network in order to check its consistency and live consistency. The generated state space turns out to be much smaller than that of the component network because we have abstracted away from the internal activity of components. Table 6.2 shows the size of state spaces of the two networks, generated by our Moses tool.

We have ensured the consistency and live consistency of the formed IA network using our implemented analysis tools. From Section 3.3.3.3, we know our component-based design of the Production Cell is consistent and free from deadlock. In addition, from Table 6.2, it is clear that this compositional verification approach results in significant reduction on memory usage and time consumption in proving the basic properties.

6.3.2 Verifying Safety Properties

In addition to the basic properties, there are also many safety requirements in the Production Cell case study [88].

To specify the safety requirements, it is essential to make an assumption on the physical world. That is, the control software of an actuator described by our models always runs faster than the actuator itself, so that the control software is able to stop the actuator once the stop condition is satisfied and before the actuator overshoots.

In the following, we classify these requirements into three categories: component invariants, system boundedness and system-wide safety. We then demonstrate how to specify and prove each of them from the previously built state spaces.

6.3.2.1 Component Invariants

Component invariants are predicates involving only places and transitions in a single component. These include requirements related to design consistency, such as the complementarity and boundedness of places, and the restrictions on machine movement.

For instance, the feed/deposit belt design requires the complementarity of places, e.g. "$cmdStart$" vs. "$cmdStop$" and "$lightBarrierTrue$" vs. "$lightBarrierFalse$". We can express this requirement using ELAN as follows:

$$\begin{aligned}&(\#cmdStart + \#cmdStop = 1)\\ \wedge\ &(\#lightBarrierTrue + \#lightBarrierFalse = 1).\end{aligned}$$

Also, it is required that all places in a component can have at most one token. It can be formulated as the following, assuming $PlaceNames$ contains the names of all places in a component:

$$\forall pn \in PlaceNames, \#pn \leq 1.$$

Furthermore, a movement restriction states that the robot must not be rotated clockwise if its first arm points towards the table, or rotated anticlockwise if its first arm points towards the press. As we have assumed that the control software is always fast enough to stop the actuator on time, this requirement can only be violated when transition $ldRotStart1$ becomes enabled while robot is still at the loading (or pickup) angle of the first arm, or when any of transitions $ldRotStart2$, $uldRotStart2$ and $uldRotStart1$ becomes enabled while the robot is at the unloading (or release) angle of the first arm. Therefore, the ELAN expression for the above requirement is as follows:

$$\begin{aligned}&\neg\,(atLdAngle1 \wedge ldRotStart1)\\ \wedge\ &\neg\,(atUldAngle1 \wedge (ldRotStart2 \vee uldRotStart2 \vee uldRotStart1)),\end{aligned}$$

where we use a transition name to denote the enabledness of the transition, and for brevity we use a place name to represent the existence of a token in the place, because all places are bounded to 1, as stated below.

As demonstrated in Section 3.3.5, the invariants that hold in the local state space of a component also hold in the system, provided the system is consistent. To prove these invariants in the system, we only need to check the local state space of the component. We have formulated and checked all component invariants in the Production Cell design (cf. Appendix A for other component invariants). They were found to be true in the respective component local state spaces. Therefore, they also hold in the component network (or system).

6.3.2.2 System Boundedness

System boundedness refers to the boundedness of all places in the system. Since the boundedness of a single place is a component invariant, we can simply prove it in the component local state space. Hence we can prove the boundedness of all places in our design. Following this approach, we have proved that the flattened net of the whole system is a 1-safe Petri net.

6.3.2.3 System-Wide Properties

System-wide safety properties are conjunctions of property clauses involving places and transitions in two or more components. In particular, these include requirements for avoiding machine collisions, preventing blanks from being dropped outside the safe area, and ensuring sufficient distance between blanks [88]. Example requirements and their specifications in ELAN are given below, where as above we simply use a place name to represent the existence of a token in the place.

1. The press may only close when no robot arm is positioned inside it.

$$
\begin{aligned}
& press.((cmdStop \wedge atUpper) \vee (cmdGoUp \wedge (fgRun \vee fgRs))) \\
\Longrightarrow\ & (\neg\, (robot.atUldAngle1 \wedge arm1.atReleaseExt) \\
& \wedge \neg\, (robot.atLdAngle2 \wedge arm2.atPickupExt))
\end{aligned}
$$

In the above, $press.cmdStop$ represents that the stop command is the last received control command by the motor of the press. In other word, the lower plate of the press is stopped. Further, $press.atUpper$ indicates that the lower plate is at the upper position. Hence $press.(cmdStop \wedge atUpper)$ states that the press is closed. In addition, $press.cmdGoUp$ means that the motor has received a "going-up" command and is moving the lower plate upwards. Further, $press.fgRun$ and $press.fgRs$ indicate that the forging procedure is in process, where fg stands for "forge". Thus $press.(cmdGoUp \wedge (fgRun \vee fgRs))$ states that the press is closing. Finally, the second/third line requires that arm1/2 is not positioned inside the press.

2. The feed belt may only convey a blank through its light barrier, if the table is stopped and in the loading position.

$$
\begin{aligned}
& feedBelt.(cmdStart \wedge (dlvrRun \vee dlvrRs)\ ) \\
\Longrightarrow\ & (\ table.atLdAngle \wedge table.atBottom \\
& \wedge table.cmdStopH \wedge table.cmdStopV\ )
\end{aligned}
$$

The first line indicates that a blank is being delivered. The second line states that the table is in the loading position, where "$table.atLdAngle$" and "$table.atBottom$" represents that the table is at its loading angle and its bottom position, respectively. Finally, the last line suggests that both the motors in charge of horizontal rotation and vertical movement of the table are stopped, where "$table.cmdStopH$" and "$table.cmdStopV$" are the stop commands for the motors, respectively.

3. A new blank may only be put on the deposit belt, if the former blank has arrived at the end of the deposit belt.

In the following, we formulate a stronger requirement stating that $arm2$ may only put a blank on the deposit belt, if the deposit belt is ready to accept blanks.

$$(robot.atUldAngle2 \wedge arm2.atReleaseExt \wedge arm2.release)$$
$$\Longrightarrow depositBelt.wait4Ld$$

4. The robot may only rotate if both arms are retracted.

$$(robot.cmdRotClockwise \vee robot.cmdRotAnticlockwise)$$
$$\Longrightarrow (arm1.atRetractExt \wedge arm2.atRetractExt)$$

To verify such system-wide properties, we apply the compositional verification method developed in Section 3.3.5. Clearly, all the above properties can be expressed in terms of property clauses or their conjunctions. Therefore, using our analysis tools described in Chapter 5, we have successfully proved these properties for our Production Cell design. As noted in Section 5.3.5, the states to be examined include both those of the IA network and those of all involved components. Table 6.3 shows the number of states examined for these verification tasks.

Req. No.	Number of states examined			
1	$74_{(robot)}$,	$60_{(arm)}$,	$36_{(press)}$,	$692_{(IAN)}$
2	$21_{(feedBelt)}$,	$44_{(table)}$,		$692_{(IAN)}$
3	$74_{(robot)}$,	$60_{(arm)}$,	$44_{(depositBelt)}$,	$692_{(IAN)}$
4	$74_{(robot)}$,	$60_{(arm)}$,		$692_{(IAN)}$

Table 6.3: Results for verifying system-wide properties

Using the same method, we have also proved the other safety requirements listed in [88] on our design. An extensive coverage of those properties can be found in Appendix A. Our experiments show that the maximum state space handled for checking each requirement turns out to be that of the derived IA network, which consists of 692 states. Clearly, compared with a monolithic checking on the component network which needs to store

$2,068,884$ states, our compositional method has achieved a significant reduction on the size of the state space required for verifying each system-wide property.

6.4 Summary and Related Work

In this chapter, we have applied the component-based design and verification methodology developed earlier to the Production Cell case study. This uses interface automata (IAs) to capture the communication protocol for each component, and employs a divide-and-conquer approach to the verification of the consistency, deadlock freedom and safety requirements. It was shown that this compositional verification approach resulted in approximately three orders of magnitude improvement in the size of the required state space.

In the literature, there are many studies of the Production Cell, *e.g.* [24, 31, 88, 89, 138, 140]. Among them, the closest to our approach is the work of [88, 89], where a detailed model of the Production Cell is found. Our design builds on this model but segments it into 7 loosely-coupled reusable components. Also, that work differs from ours in the employed verification method. There, the verification was directly conducted on the global state space of the system with the help of reduction techniques such as stubborn set methods. In our work, the costly construction of the global state space is avoided. Instead, with the help of interface automata, system properties are proved by checking a number of small state spaces.

In [140], Lilius and Paltor presented a design and verification approach to the Production Cell on the basis of UML. There, components are modelled as UML Statecharts and the verification was conducted using the vUML tool [139] which invokes the SPIN tool [96] for executing the model checking task. As in [88, 89], this approach employs reduction techniques in order to explore all possible states of the system. This, however, trades time for memory. In contrast, our approach requires much less time and memory due to the smaller state spaces that need to be handled. Furthermore, our approach is not dedicated to a particular modelling language, but accommodates various notations for modelling components, including UML Statecharts.

In [24], Börger and Mearelli presented a design and verification approach to the Production Cell on the basis of ASMs [82]. This approach uses stepwise refinement of

ASMs for modular development of the Production Cell system, and uses explicit contextual assumptions of components to assist the modular proving of system properties. In contrast to our approach, all properties including the refinement relationships of components are proved by hand in this approach.

In [31], Cheung and Kramer applied a compositional minimisation approach to the verification of the Production Cell. The size of the state spaces that need to be constructed is approximately the same as our approach. However, due to the lack of knowledge about the size of state space of their original component-based system, we cannot make a quantitative comparison with their approach.

In addition, [138] contains a collection of other contributions and a detailed comparative survey on this case study. However, no approach to the verification of the Production Cell contained in [138] is both compositional and automatic.

7 Conclusions

No single specification or verification method is able to solve all classes of problems. To cope with industrial-sized applications, we need not only a diversity of modelling languages and analysis techniques specialised and optimised for various domains, but also the ability to use them in combination.

The work presented in this thesis has considered these fundamental issues. More specifically, in order to cope with the growing heterogeneity of computer-based systems, it has concentrated on developing techniques to support the use of a combination of modelling languages, especially visual languages, for system specification. Also, in order to tackle the main obstacles of model checking and make it more accessible to and usable by practising engineers, this work has focussed on providing lightweight but effective methods and tools to alleviate the state space explosion problem in model checking.

7.1 Contributions

In this thesis, we have accomplished several tasks. We have constructed a formal semantic base for heterogeneous systems in terms of interconnected discrete-event components. This allows us to conduct research on heterogeneous systems along two lines: semantic definition approaches and verification techniques.

To define an operational semantics for heterogeneous systems, we have presented an interpretation approach, which extends the work of [65, 110] and constructs component interpreters with built-in facilities to enable exhaustive state space analysis. In essence, a component interpreter is a modelling-language-specific interpreter parameterized by a component model. It is responsible for defining states and transitions of the component, defining the enabled steps of components at each state, and firing a specified enabled step or receiving an input when required, according to the semantics of the modelling language. By fulfilling these obligations, the component interpreter allows an analysis tool to conduct an exhaustive exploration of state space of the component and a system comprising it. Based on such interpreters, we have implemented a simple state space analysis tool which gives heterogeneous systems an operational semantics in terms of DECs. As such, we have developed an extensible underlying framework for the formal verification of heterogeneous systems. It imposes an explicit behavioural contract between interpreters and analysis tools, and enables a combination of languages to be used to model complex systems and a variety of model checking techniques to be used for their verification.

In addition, we have developed a compositional approach to the verification of heterogeneous systems, which combats the state space explosion problem using the principle of "divide and conquer". This approach describes the assumed and guaranteed behaviour of each component in a single IA, and follows three steps to prove the basic properties of a heterogeneous system such as consistency and deadlock freedom. Firstly, this approach adopts an optimistic view of the environment and checks the fulfillment of behavioural guarantees by each component, assuming its behavioural assumptions are all satisfied by the environment. Next, it checks the satisfaction of the behavioural assumptions of all components in the derived IA network, by verifying the assumptions of each component against the behavioural guarantees of the other components. Finally, this approach claims the system is consistent and free from deadlock, if no violation is detected.

Furthermore, in this compositional approach, verifying safety properties is also possible, despite the abstraction from components to IAs. Three kinds of safety properties, including local properties of components, system boundedness and system-wide properties, are distinguished and handled using methods with different computational complexities. Local properties of a component are determined solely in its local state space, and system boudedness are determined by checking the boundedness of every component in its local state space. In contrast, verifying a system-wide property is more complex. It involves deducing from the derived IA network a superset of actual state combinations of the relevant components, and claiming the satisfaction of the property if it is not violated by elements in this superset.

Moreover, we have implemented this compositional approach in Java-based verification tools in the context of the Moses tool suite [65], and have applied the tools to the verification of a non-trivial system: the Production Cell [138].

In summary, specific contributions of the thesis include

- the formulation of a semantic base for heterogeneous components (Chapter 3). This is built on discrete-event components which are sufficiently general to accommodate a class of pragmatic specification languages, in particular, graph-like visual notations;
- the formulation of conformance relationships between heterogeneous components and IAs, together with the definitions of basic and safety properties of heterogeneous systems, adopting a semantics based on interconnected DECs (Chapter 3);
- the development and theoretical justifications of practical methods for checking the conformance of heterogeneous components and compositional methods for verifying the basic and safety properties of heterogeneous systems (Chapter 3);
- the presentation of a semantic interpretation approach to heterogeneous systems (Chapter 4). This approach avoids a naive direct mapping from components into DECs. Instead, it takes a practical two-stage process based on language-specific interpreters and analysis tools to give an operational semantics to components in terms of their reachable states and transitions in a given context;
- the formulation of an explicit behavioural contract between language-specific interpreters and analysis tools (Chapter 4). This includes the definition of analysis variables of interpreters exposing the state and transition information of components as well as the expected behaviour of analysis tools by interpreters. This formulation enables the independent development of interpreters and analysis tools;

- the presentation of a formal operational semantics for UML statecharts, a semi-formal but very important language in UML, adopting the proposed semantic interpretation approach (Chapter 4);
- the development of a simple state space analysis tool for heterogeneous systems, together with the introduction of a property specification language (Chapter 5). This tool is implemented using Java in the context of the Moses tool suite, abiding by the above-mentioned behavioural contract. It provides an operational semantics to heterogeneous systems in terms of DECs and supports a monolithic approach to their model checking;
- the implementation of the above-mentioned compositional verification methods in the context of the Moses tool suite (Chapter 5). This implementation uses the previously mentioned simple analysis tool as a base to check the conformance of components, the consistency of IA networks, and local safety properties of components. Furthermore, it employs a backward search for verifying the live consistency of IA networks and the live conformance of components and open systems. In addition, it utilises a special algorithm to determine system-wide safety properties;
- the compositional verification of a non-trivial system: the Production Cell (Chapter 6). Using our developed tools, the 21 safety properties of the system listed in [88] have been successfully proved. It was shown that approximately three orders of magnitude improvement in the size of the required state space was achieved.

7.2 Future Work

The work presented in this thesis establishes a framework for the formal specification and verification of component-based heterogeneous systems. It also contributes to the state-of-the-art of model checking in the area of compositional verification. Along these lines, a number of directions can be further explored to extend and improve this work.

Firstly, the proposed verification framework for heterogeneous systems may be extended to incorporate more specification languages and model checking techniques.

- In this thesis, two languages, Petri nets and UML statecharts, have been incorporated for the specification of heterogeneous systems. More graph-like languages may be added in the future to provide more flexibility for the specification of complex systems. Furthermore, it is believed that textual languages able to represent discrete-event systems can be integrated into this framework. Future work may focus on a class of textual languages, identify a suitable abstract syntactic model for them, and then apply the proposed semantic interpretation approach to their interpreter specification. Ultimately, these languages may be used to specify heterogeneous components. In addition, research may be conducted on the possibility and means to extend this framework to support other kinds of specification languages, *e.g.* the Specification and Description Language (SDL) [106], Message Sequence Charts (MSCs) [107], and UML Collaboration diagrams [159]. Depending on their particular characteristics, this may require certain extensions to the semantic base underlying this framework.

- In this thesis, two kinds of analysis tools have been integrated for the verification of heterogeneous systems. These include a simple state space analysis tool and a compositional verification tool. It was shown that developing these analysis tools is independent of particular modelling languages, thanks to the explicit behavioural contract imposed in Chapter 4. In the future, other compositional verification and model checking techniques, *e.g.* on-the-fly model checking and state space reduction, may be included to create a more flexible verification approach. Also, future research may be focused on issues about their use in combination with our compositional verification approach.

The research on extending the specification and verification capabilities of our proposed verification framework would help maximise the potential of this framework and allow its use by practising engineers to cope with realistic systems.

Secondly, our compositional verification approach provides an over-approximation for verifying system properties, due to the introduced abstraction between components and interface automata. That is to say, it may incorrectly report violations for some properties. Methods and techniques (preferably with automated tool support) are needed to be able to identify those false negatives and (maybe heuristically) refine IAs to capture more detailed behaviours of components. In particular, the approach presented by Clarke *et al.* [35] may be adopted. Moverover, to further demonstrate the power or identify the weaknesses of our approach, more extensive case studies and comparisons with other model checking approaches are needed. In addition, path-based safety properties and liveness are not considered in this thesis. They may also be considered in the future work.

A The Production Cell: Further Detail

In Chapter 6, we have presented some component models for the Prodcution Cell [138], including the feed belt, the deposit belt, the arms and the robot component. We have also illustrated a compositional approach to the verification of the cell using some example safety properties. In this section, we provide the remaining component models including the table, the press and the crane. In addition, we specify all the other safety properties listed by Heiner and Deussen [88], and present the verification results of these properties.

A.1 Remaining Component Models

A.1.1 Elevating Rotary Table

The main job of the table is to bridge both the horizontal and vertical gaps between the feed belt and the first robot arm. It is therefore fitted with two motors in charge of the horizontal rotation and the vertical movement, respectively. In the model of the table shown in Figure A.1, their control commands are modelled by two groups of complementary places: $\{cmdStopH, cmdRotClockwise, cmdRotAntiClockwise\}$ and $\{cmdStopV, cmdGoUp, cmdGoDown\}$, respectively. In addition, the table has sensors to detect its current position. These include two (boolean valued) switches used to indicate if the table is at the top and bottom positions, and an analog potentiometer used to detect the rotation angle. In Figure A.1, these are modelled as two groups of complementary places:

$\{atTop, atBottom\}$ and $\{atLdAngle, atUldAngle\}$. Here, as usual, we have simplified the reading of the potentiometer into two discrete values.

Initially, the table is at its loading position (*i.e.* at both the loading angle and the bottom position) with a loaded blank, and both motors have stopped. Next, the table will go to its unloading angle and the top position in order for the first robot arm to pick up the loaded blank. This involves firing all the transitions at the top line of Figure A.1. The table will then send an $oFull$ message to notify the first arm to pick up. When the arm finishes, *i.e.* when an $oFree$ message is received, the table will move back to its loading position, firing the transitions at the bottom line of Figure A.1. Next, it will release the input lock by sending a token via $iFree$ to the feed belt, and wait for a new blank to be loaded.

A.1.2 Press

The press forges blanks by pressing its lower plate against its upper one. It is powered by a motor. In its component model shown in Figure A.2, the control commands of this motor are modelled by three complementary places: *cmdStop*, *cmdGoUp* and *cmdGoDown*. In addition, the sensors, *i.e.* three switches, are modelled by three complementary places: $atUpper$, $atMiddle$ and $atLower$.

Initially, the press is at the middle position with a loaded blank. Next, it will go to forge the blank by lifting up its lower plate, *i.e.* firing the transitions $forgeStart$, $fgWait$, and $fgStop$ at the top line of Figure A.2. After that, the press will go to the lower position and notify the second robot arm to pick up the forged blank, firing $lowerStart$, $lwWait$, $lwStop$, and $oUnlock$. When the arm finishes, *i.e.* when an $oFree$ message is received, the press will move back to the middle position by firing the transitions at the bottom line of Figure A.2. It will then release the input lock by sending a token via $iFree$ to the first robot arm, and wait for a new blank to be loaded.

A.1.3 Crane

The main job of the crane is to move blanks at the end of the deposit belt back to the feed belt. It is powered by two motors in charge of the horizontal and vertical movement of its electromagnet gripper. In its component model shown in Figure A.3, the

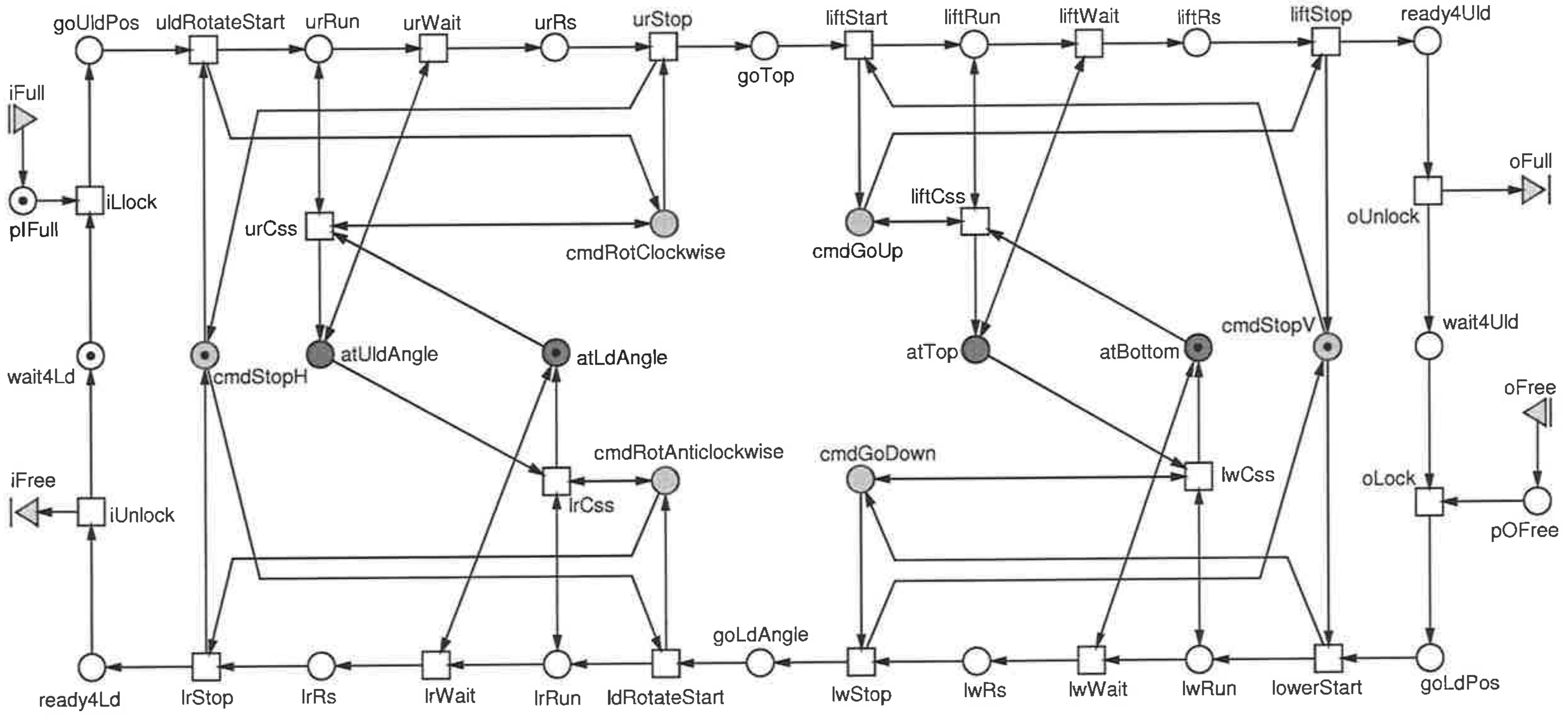


Figure A.1: Component model for the table

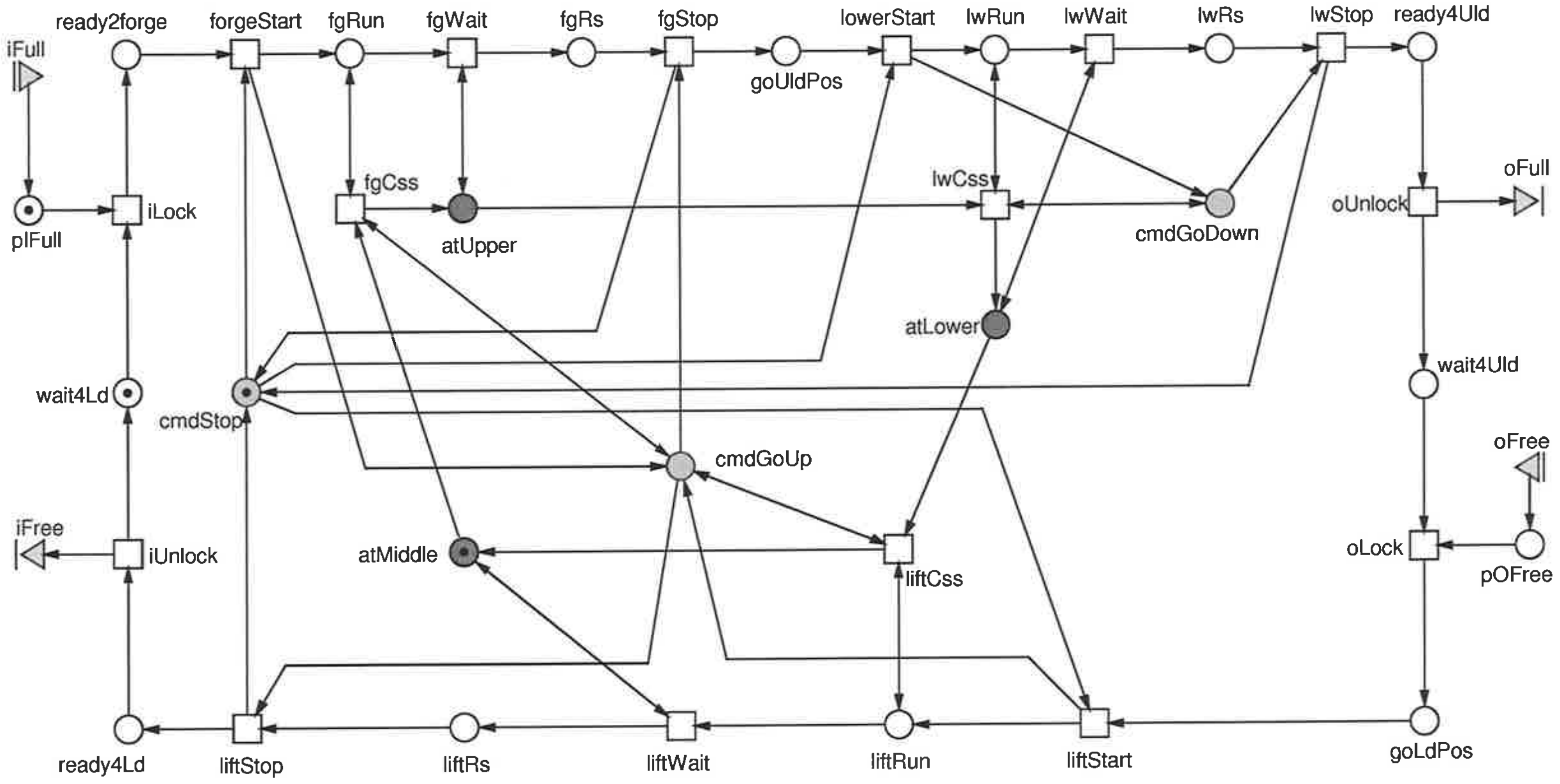


Figure A.2: Component model for the press

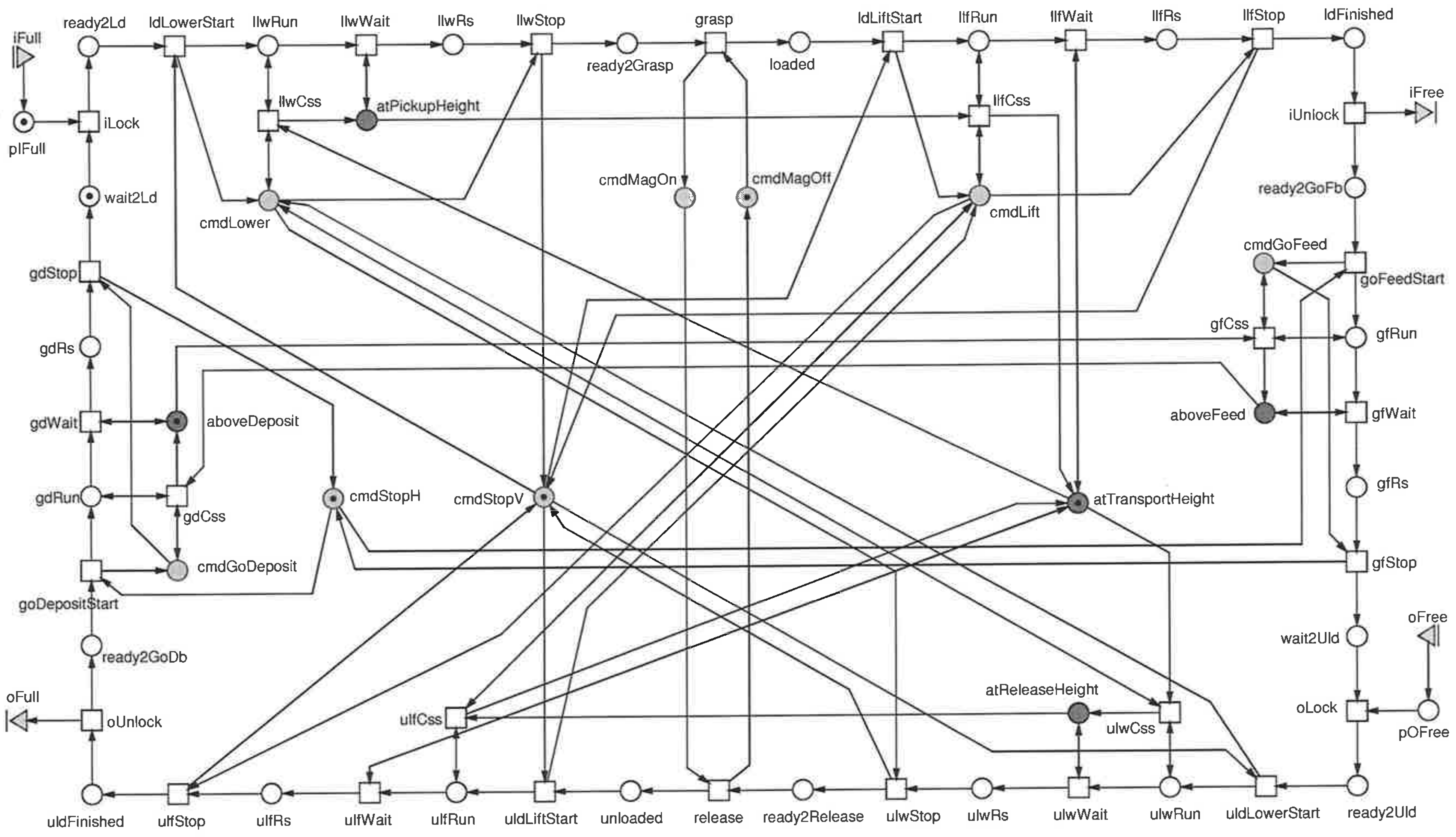


Figure A.3: Component model for the crane

control commands of these motors are modelled by two groups of complementary places: $\{cmdStopH, cmdGoDeposit, cmdGoFeed\}$ and $\{cmdStopV, cmdLift, cmdLower\}$. Further, the blank grasp and release by the electromagnet gripper is described by two complementary places: *cmdMagOn* and *cmdMagOff*. In addition, the position of the gripper is described by two groups of complementary places: $\{aboveDeposit, aboveFeed\}$ and $\{atPickupHeight, atReleaseHeight, atTransportHeight\}$.

Initially, the gripper is above the deposit belt and stopped at the transportation height. As a blank is initially at the end of the deposit belt, the crane will lower its gripper to pick up the blank after locking its input region. This involves firing transitions $ldLowerStart$, $llwWait$, and $llwStop$. After that, the crane will grasp the blank by firing $grasp$ and lift up the gripper back to the transportation height by firing $ldLiftStart$, $llfWait$, and $llfStop$. All these transitions are shown at the top row of Figure A.3. Next, the crane unlocks its input region and then goes to the feed belt by firing transitions at the right column of Figure A.3. At that time, if its output region is available, viz. $oFree$ has been received (meaning that the feed belt is ready for loading), the crane will follow a similar procedure to release the blank by firing transitions at the bottom row. Finally, it will release its output region by $oFull$ and go back to the deposit belt for a new blank.

A.2 Compositional Verification

In this section, we give the full detail about the important safety properties of the Production Cell listed in [88], including their specification and verification.

A.2.1 Design Consistency

This includes complementarity and boundedness of places. As noted, such a property is a component invariant and can be proved merely in the local state space of the component. Since the boundedness of places has been considered in Section 6.3.2.1, in the following, we only discuss properties involving the complementarity of places. Table A.1 lists these properties for each component. Using our Moses tool, we have successfully proved all these in their respective component local state spaces.

Component	Invariants
feed/deposit belt	Cf. Section 6.3.2.1
table	$\#cmdStopH + \#cmdRotClockwise + \#cmdRotAnticlockwise = 1$
	$\#cmdStopV + \#cmdGoUp + \#cmdGoDown = 1$
	$\#atLdAngle + \#atUldAngle = 1$
	$\#atTop + \#atBottom = 1$
robot	$\#cmdRotStop + \#cmdRotClockwise + \#cmdRotAnticlockwise = 1$
	$\#atLdAngle1 + \#atLdAngle2 + \#atLdAngle2 + \#atUldAngle1 = 1$
arm1/arm2	$\#cmdStop + \#cmdGoForward + \#cmdGoBackward = 1$
	$\#cmdMagOn + \#cmdMagOff = 1$
	$\#atRetractExt + \#atPickupExt + \#atReleaseExt = 1$
press	$\#cmdStop + \#cmdGoUp + \#cmdGoDown = 1$
	$\#atLower + \#atMiddle + \#atUpper = 1$
crane	$\#cmdStopH + \#cmdGoDeposit + \#cmdGoFeed = 1$
	$\#cmdStopV + \#cmdLift + \#cmdLower = 1$
	$\#cmdMagOn + \#cmdMagOff = 1$
	$\#atPickupHeight + \#atTransportHeight + \#atReleaseheight = 1$
	$\#aboveDeposit + \#aboveFeed = 1$

Table A.1: Proven component invariants

A.2.2 Restrictions of Machine Movement

As stated previously, all the 7 restrictions on machine movement are component invariants and can also be proved by checking only the component local state spaces. Note that in specifying such properties, we have assumed that the control software is fast enough to stop the actuator on time. Thus a restriction of machine movement can only be violated due to the reception of an illegal command, *e.g.* command $cmdRotClockwise$ while place $atLdAngle1$ of the robot is true.

Requirement 1. The robot must not be rotated clockwise if its first arm points towards the table, or rotated anticlockwise if its first arm points towards the press.

Cf. Section 6.3.2.1.

Requirement 2. Both arms of the robot must not be retracted less than necessary for passing the press, or extended more than necessary for picking up a blank from the press.

$$\begin{aligned} &\neg\,(atRetractExt \wedge (ldRetractStart \vee uldRetractStart)) \\ \wedge\ &\neg\,(atPickupExt \wedge (ldExtendStart \vee uldExtendStart)) \\ \wedge\ &\neg\,(atReleaseExt \wedge (ldExtendStart \vee uldExtendStart)) \end{aligned}$$

Requirement 3. The lower plate of the press must not be moved downward if it is in the bottom position, or moved upward if it is in the top position.

$$\begin{aligned} &\neg\,(atBottom \wedge lowerStart) \\ \wedge\ &\neg\,(atTop \wedge (forgeStart \vee liftStart)) \end{aligned}$$

Requirement 4. The table must not be moved downward if it is in the bottom position, or moved upward if it is in the top position.

$$\begin{aligned} &\neg\,(atBottom \wedge lowerStart) \\ \wedge\ &\neg\,(atTop \wedge liftStart) \end{aligned}$$

Requirement 5. The table must not be rotated clockwise if it is at the unloading angle, or rotated anticlockwise if it is at the loading angle.

$$\begin{aligned} &\neg\,(atUldAngle \wedge uldRotateStart) \\ \wedge &\neg\,(atLdAngle \wedge ldRotateStart) \end{aligned}$$

Requirement 6. The crane may only move towards the deposit belt if positioned above the feed belt and only move towards the feed belt if positioned above the deposit belt.

$$\begin{aligned}&\neg\,(aboveFeedBelt \wedge goFeedStart)\\ \wedge\neg\,&(aboveDepositBelt \wedge goDepositStart)\end{aligned}$$

Requirement 7. The gripper of the crane must not be moved downward if it is in the positions required for picking up a blank from the deposit belt and for releasing a blank to the feed belt. Also, it must not be moved above the position required for transportation.

$$\begin{aligned}&\neg\,(atPickupHeight \wedge (ldLowerStart \vee uldLowerStart))\\ \wedge\neg\,&(atReleaseHeight \wedge (ldLowerStart \vee uldLowerStart))\\ \wedge\neg\,&(atTransportHeight \wedge (ldLiftStart \vee uldLiftStart))\end{aligned}$$

A.2.3 Avoidance of Machine Collisions

These include both component invariants such as Requirement 10–11, and system-wide properties such as Requirement 8–9.

Requirement 8. The press may only close when no robot arm is positioned inside it.

Cf. Section 6.3.2.3.

Requirement 9. The robot may only rotate if both arms are retracted.

Cf. Section 6.3.2.3.

Requirement 10. The travelling crane is not allowed to knock against a belt laterally.

We check a stronger condition, that is, when performing a horizontal translation, the crane gripper must have been lifted to the transportation height.

$$crane.(cmdGoFeed \vee cmdGoDeposit) \implies crane.atTransHeight$$

Requirement 11. The crane must not knock against a belt from above.

Due to the assumptions we made on the physical world, the specification of this requirement is the same as Requirement 7.

A.2.4 No Blanks Dropped Outside Safe Areas

All these requirements except Requirement 14 involves places and transitions in two or more (usually neighboring) components. They are thus system-wide properties and should be verified using the method proposed in Section 3.3.5.

Requirement 12. The magnet of the first robot arm may only be deactivated, if the arm points towards the press and is extended such that the gripper is within the press.

$$arm1.release \implies (arm1.atReleaseExt \wedge robot.atUldAngle1)$$

Requirement 13. The magnet of the second robot arm may only be deactivated, if the magnet is above the deposit belt.

$$arm2.release \implies (arm2.atReleaseExt \wedge robot.atUldAngle2)$$

Requirement 14. The magnet of the crane may only be deactivated, if the crane's gripper is above the feed belt and sufficiently close to it.

$$crane.release \implies crane.(aboveFeed \wedge atReleaseHeight)$$

Requirement 15. The feed belt may only convey a blank through its light barrier, if the table is stopped in the loading position.

Cf. Section 6.3.2.3.

Requirement 16. The deposit belt must be stopped after a blank has passed the light barrier at its end, and may only be started again after the crane has picked up the blank.

For the first half, we specify a weaker requirement in the following, stating that the belt is stopped at all possible states after blank delivery and before the blank is picked up by the crane.

$$\begin{aligned} &depositBelt.(delivered \vee ready4Ld \vee wait4Ld \vee wait2Tran) \\ &\implies depositBelt.cmdStop \end{aligned}$$

In addition, we express a weaker requirement for the other half as follows, stating that the belt cannot be started when the crane is picking up the blank:

$$\neg\,(crane.grasp \wedge depositBelt.cmdStart)$$

A.2.5 Insurance of a Sufficient Distance between Blanks

All these requirements are system-wide properties to be verified using the method proposed in Section 3.3.5.

Requirement 17. A new blank may only be put on the feed belt, if the former blank has arrived at the end of the feed belt.

We formulate a stronger requirement ensuring that the feed belt is always ready when the crane releases a loaded blank onto the belt.

$$crane.(aboveFeedBelt \wedge release) \implies feedBelt.wait4Ld$$

Requirement 18. A new blank may only be put on the deposit belt, if the former blank has arrived at the end of the deposit belt.

Cf. Section 6.3.2.3.

Requirement 19. Blanks may not be put on the table, if the table is already loaded.

We rephrase this requirement as “Blanks may only be put on the table, if the table is not loaded.”

$$feedBelt.(cmdStart \wedge (dlvrRun \vee dlvrRs)) \implies table.ready4Ld$$

Requirement 20. Blanks may not be put on the press, if the press is already loaded.

We rephrase this requirement as “Blanks may only be put on the press, if the press is not loaded.”

$$arm1.release \implies press.ready4Ld$$

Next, we formulate Requirement 21 in [88] as the conjunction of the following two requirements (*i.e.* 21a and 21b).

Requirement 21a. The robot must only be rotated clockwise if both arms are unloaded.

$$\begin{aligned} &robot.cmdRotClockwise \\ &\implies (arm1.cmdMagOff \wedge arm2.cmdMagOff) \end{aligned}$$

Requirement 21b. The table, if loaded, must not enter its unloading position, if arm 1 is already loaded and still in its loading position (otherwise the two blanks may collide).

$$\begin{aligned} &(robot.atLdAngle1 \wedge arm1.cmdMagOn \wedge arm1.atPickupExt) \\ \implies &\neg\, table.(cmdGoUp \wedge atUldAngle) \end{aligned}$$

A.2.6 Summary of the Verification Results

For Requirement 1–7, 10, 11 and 14, we have checked and successfully proved them in the local state space of each involved component. According to Theorem 9, they also hold in the the component-based system of Production Cell. Our experiments show that checking Requirement 6, 7, 10, 11 and 14 requires handling the largest set of states, more specifically, all the 80 states in the local state space of the crane component.

For the other requirements, we employed the compositional approach proposed in Chapter 3 and have also successfully proved them in the component-based system, using our analysis tools developed in Chapter 5. As already noted in Chapter 6, verifying each requirement involves checking the states of the derived IA network and those of all the components relevant to the property. As a result, the maximum state space handled turns out to be that of the derived IA network. As shown in Table 6.3, this consists of 692 states. Clearly, this is much smaller than the original state space of the component-based system, which consists of $2,068,884$ states. Therefore, approximately three orders of magnitude reduction on memory requirement is achieved.

Bibliography

[1] The Design/CPN tool. https://www.daimi.au.dk/designCPN/.

[2] The Moses project. Computer Engineering and Communications Laboratory, Swiss Federal Institute of Technology (ETH) Zurich. http://www.tik.ee.ethz.ch/˜moses/.

[3] The Moses tool suite. https://sourceforge.net/projects/mosestoolsuite/.

[4] The UPPAAL tool. http://www.uppaal.com/.

[5] *Proceedings of the 23rd International Conference on Software Engeneering (ICSE'01)*, Toronto, Canada, 2001. IEEE Computer Society.

[6] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[7] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[8] P. Alexander and C. Kong. Rosetta: Semantic support for model-centered systems-level design. *IEEE Computer*, 34(11):64–70, November 2001.

[9] R. Alur and T. A. Henzinger, editors. *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96)*, LNCS 1102, 1996.

[10] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design: An International Journal*, 15(1):7–48, July 1999.

[11] R. Alur, T. A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996.

[12] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *[101]*, pages 521–525, 1998.

[13] A. Arnold. *Finite transition systems - Semantics of communicating processes*. Prentice Hall, 1994.

[14] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[15] R. Bastide, O. Sy, and P. Palanque. Formal specification and prototyping of CORBA systems. In R. Guerraoui, editor, *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, LNCS 1628, pages 474–494. Springer, 1999.

[16] I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *[101]*, pages 184–194, 1998.

[17] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property Preserving Simulations. In G.V. Bochmann and D.K. Probst, editors, *Proceedings of the 4th Workshop on Computer Aided Verification (CAV'92)*, July 1992.

[18] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *[101]*, pages 319–331, 1998.

[19] S. S. Bhattacharyya, E. Cheong, J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in Java. Technical Memorandum UCB/ERL M03/27, EECS, University of California, Berkeley, July 2003.

[20] A. D. Bimbo, L. Rella, and E. Vicario. Visual specification of branching time temporal logic. In *Proceedings of the IEEE Symposium on Visual Languages (VL'95)*, pages 61–68, 1995.

[21] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, Henny Sipma, and T. E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *[9]*, pages 415–418, 1996.

[22] L. Blair and G. Blair. Composition in multiparadigm specification techniques. In *Proceedings of the IFIF TC6/WG6.1 3rd International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, pages 401–417. Kluwer, 1999.

[23] T. Bolognesi and E. Brinksma. Introduction to ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[24] E. Börger and L. Mearelli. Integrating ASMs into the software development life cycle. *Journal of Universal Computer Science*, 3(5):603–665, 1997.

[25] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[26] A. W. Brown and K. C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, 1998.

[27] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transaction on Computers*, 35(6), 1986.

[28] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. Technical Report CS-93-211, School of Computer Science, Carnegie Mellon University, July 1993.

[29] C. Canal, L. Fuentes, E. Pimentel, J. M. Troya, and A. Vallecillo. Extending CORBA interfaces with protocols. *Computer Journal*, 44(5):448–462, 2001.

[30] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, 1998.

[31] S. C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 5(4):334–377, October 1996.

[32] S. C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 8(1):49–78, January 1999.

[33] S. Christensen, L. M. Kristensen, and T. Mailund. A sweep-line method for state space exploration. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, LNCS 2031, pages 450–464, 2001.

[34] C. D. T. Cicalese and S. Rotenstreich. Behavioral specification of distributed software component interfaces. *IEEE Computer*, 32(7):46–53, July 1999.

[35] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *[63]*, 2000.

[36] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, LNCS 131, pages 52–71. Springer-Verlag, May 1981.

[37] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[38] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis, editor, *Proceedings of The 5th Workshop on Computer Aided Verification (CAV'93)*, pages 450–462, June/July 1993.

[39] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, 1999.

[40] E. M. Clarke and R. P. Kurshan, editors. *Proceedings of the 2nd Workshop on Computer-Aided Verification (CAV'90)*, LNCS 531, Berlin, Germany, June 1990. Springer.

[41] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science*, pages 353–362, Pacific Grove, California, June 1989.

[42] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. In *Proceedings of the 9th International Symposium on Computer Hardware Description Languages and their Applications*, pages 281–295, June 1989.

[43] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future. *ACM Computing Survey*, 28(4), December 1996.

[44] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[45] M. A. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. *LNCS 1427*, pages 293–304, 1998.

[46] D. Compare, P. Inverardi, and A. L. Wolf. Uncovering architectural mismatch in component behavior. *Science of Computer Programming*, 33(2):101–131, 1999.

[47] I. Crnkovic and M. Larsson, editors. *Building Reliable Component-Based Software Systems.* Artech House, 2002.

[48] R. M. Cubert and P. A. Fishwick. MOOSE: An object-oriented multimodeling and simulation application framework. *Simulation*, June 1997.

[49] D. Dams, R. Gerth, and O. Grumberg. Generation of reduced models for checking fragments of CTL. In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93)*, LNCS 697, pages 479–490, Elounda, Greece, 1993. Springer-Verlag.

[50] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.

[51] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, Trento, Italy, July 1999. Springer-Verlag.

[52] N. A. Day. *A Framework for Multi-Notation, Model-Oriented Requirements Analysis*. PhD thesis, Department of Computer Science, University of British Columbia, Vancouver, B.C., Canada, October 1998.

[53] N. A. Day and J. J. Joyce. A framework for multi-notation requirements specification and analysis. In P. Hsia, B. Cheng, and D. Weiss, editors, *Proceedings of the IEEE International Conference on Requirements Engineering (ICRE'00)*, pages 39–48. IEEE Computer Society, June 2000.

[54] L. de Alfaro and T. A. Henzinger. Interface automata. In V. Gruhn, editor, *Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE'01)*, Software Engineering Notes 26(5), pages 109–120, New York, 2001. ACM Press.

[55] J. de Lara and H. Vangheluwe. Computer aided multi-paradigm modelling to process Petri nets and statecharts. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the International Conference on Graph Transformation (ICGT'02)*, LNCS 2505, pages 239–253. Springer, 2002.

[56] G. Del Castillo. The ASM workbench — A tool environment for computer-aided analysis and validation of abstract state machine models tool demonstration. LNCS 2031, pages 578–581, 2001.

[57] D. L. Dill, editor. *Proceedings of the 6th International Conference on Computer-Aided Verification (CAV'94)*, LNCS 818. Springer-Verlag, 1994.

[58] D. L. Dill. The murϕ verification system. In *[9]*, pages 390–393, 1996.

[59] L. K. Dillon and R. E. K. Stirewalt. Lightweight analysis of operational specifications using inference graphs. In *[5]*, pages 57–70, 2001.

[60] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, Mass., 1 edition, 1999.

[61] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2), jan 2003.

[62] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9:103–131, August 1996.

[63] E. A. Emerson and A. P. Sistla, editors. *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, LNCS 1855, Chicago, IL, USA, July 2000. Springer.

[64] R. Esser. *An Object Oriented Petri Net Approach to Embedded System Design*. PhD thesis, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology (ETH) Zurich, 1996.

[65] R. Esser and J. W. Janneck. Moses - a tool suite for visual modelling of discrete-event systems. In *Proceedings of the IEEE Symposium on Visual/Multimedia Approaches to Programming and Software Engineering, Human-Centric Computing (HCC'01)*, pages 272–279, 2001.

[66] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP - a protocol validation and verification toolbox. In *[9]*, pages 437–440, 1996.

[67] A. Finkelstein, J. Kramer, and M. Goedicke. Viewpoint oriented software development. In *Proceedings of the 3rd International Workshop on Software Engineering and Its Applications*, 1990.

[68] A. C. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, August 1994.

[69] P. Fradet, D. L. Métayer, and M. Périn. Consistency checking for multiple view software architectures. In O. Nierstrasz and M. Lemoine, editors, *Proceedings of ESEC/FSE'99*, volume 1687 of *LNCS*, pages 410–428, 1999.

[70] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[71] D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, Department of Computing, Imperial College, University of London, Jan 1999.

[72] H. Giese. Contract-based component system design. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences (HICSS'00)*. IEEE Computer Society, 2000.

[73] P. Godefroid. Using partial orders to improve automatic verification methods. In *[40]*, pages 176–185, 1990.

[74] P. Godefroid, G. J. Holzmann, and D. Pirottin. State-space caching revisited. *Formal Methods in System Design*, 7(3):227–241, November 1995.

[75] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the 3rd International Conference on Computer Aided Verification (CAV'91)*, pages 332–342, 1991.

[76] M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. M. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, Boston, 1988.

[77] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *[80]*, pages 72–83. Springer Verlag, 1997.

[78] S. Graf and B. Steffen. Compositional minimization of finite state systems. In *[40]*, pages 186–196, 1990.

[79] S. Graf, B. Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Formal Aspects of Computing*, 8(5):607–616, 1996.

[80] O. Grumberg, editor. *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, LNCS 1254. Springer-Verlag, 1997.

[81] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[82] Y. Gurevich. Evolving Algebras. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 423–427, Elsevier, Amsterdam, the Netherlands, 1994.

[83] Y. Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, 2000.

[84] V. Gyuris and A. P. Sistla. On-the-fly model checking under fairness that exploits symmetry. In *[80]*, pages 232–243, 1997.

[85] D. Harel and E. Gery. Executable object modeling with statecharts. In *Proceedings of the 18th International Conference on Software Engineering*, pages 246–257. IEEE Computer Society Press / ACM Press, 1996.

[86] D. Harel and Amnon Naamad. The STATEMATE Semantics of Satecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[87] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 54–64. IEEE Computer Society Press, 1987.

[88] M. Heiner and P. Deussen. Petri net based qualitative analysis - A case study. Technical Report I-08, Brandenburg Technical University, Cottbus, December 1995. available at http://www.informatik.tu-cottbus.de/~wwwdssz/publications/papers.html.

[89] M. Heiner, P. Deussen, and J. Spranger. A case study in design and verification of manufacturing system control software with hierarchical Petri nets. *Journal of Advanced Manufacturing Technology*, 15(2):139–152, 1999.

[90] C. Heitmeyer. On the need for "practical" formal methods. *LNCS 1486*, pages 18–26, 1998.

[91] K. L. Heninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, 6(1):2–13, 1980.

[92] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *[101]*, pages 440–451, 1998.

[93] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Proceedings of the Computer Aided Design*, pages 245–253. IEEE Press, 2000.

[94] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[95] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[96] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special Issue: Formal Methods in Software Practice.

[97] G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):289–307, November 1998.

[98] Honeywell, Inc. *DOME Guide, Version 5.2.2*, 2000.

[99] Y. Hsieh and S. Levitan. Model abstraction for formal verification. Technical Report 201, Department of Electrical Engineering, University of Pittsburgh, Pittsburgh, PA, Jan 1997.

[100] A. J. Hu. *Techniques for efficient formal verification using Binary Decision Diagrams*. PhD thesis, Department of Computer Science, Stanford University, December 1995.

[101] A. J. Hu and M. Y. Vardi, editors. *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, LNCS 1427. Springer-Verlag, 1998.

[102] Institute for Software Integrated Systems, Vanderbilt University. *GME 2000 Users Manual*, April 2001. Version 1.1.

[103] P. Inverardi and S. Uchitel. Proving deadlock freedom in component-based programming. In H. Hußmann, editor, *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering (FASE'01)*, LNCS 2029, pages 60–75. Springer, 2001.

[104] P. Inverardi, A. L. Wolf, and D. Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(3):239–272, 2000.

[105] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design: An International Journal*, 9(1/2):41–75, August 1996.

[106] ITU (International Telecommunication Union), Geneva. *ITU-T Recommendation Z.100, Specification and Description Language (SDL)*, 1999.

[107] ITU (International Telecommunication Union), Geneva. *ITU-T Recommendation Z.120, Message Sequence Chart (MSC)*, 1999.

[108] M. Jackson. *Software Requirements and Specification: a lexicon of practice, principles and prejudices*. Addison-Wesley, 1995.

[109] J. W. Janneck. Graph-type Definition Language (GTDL) - Specification. Technical report, Computer Engineering and Networks Laboratory, ETH Zurich, 2000.

[110] J. W. Janneck. *Syntax and semantics of graphs - An approach to the specification of visual notations for discrete-event systems*. PhD thesis, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology (ETH) Zurich, June 2000.

[111] J. W. Janneck and R. Esser. A predicate-based approach to defining visual language syntax. In *Proceedings of the IEEE Symposium on Visual Languages and Formal Methods, Human-Centric Computing (HCC'01)*, Stresa, Italy, September 2001.

[112] J. W. Janneck and P. W. Kutter. Mapping automata - simple abstract state machines. Technical Report 49, Computer Engineering and Networks Laboratory, ETH Zurich, 1998.

[113] J. W. Janneck and M. Naedele. Modeling hierarchical and recursive structures using parametric petri nets. In *Proceedings of the High Performance Computing Symposium (HPC)*, pages 445–452, 1999.

[114] C. Jard and T. Jeron. Bounded–memory algorithms for verification on–the–fly. In K. G. Larsen and A. Skou, editors, *Proceedings of the 3rd International Conference on Computer Aided Verification (CAV'91)*, LNCS 575, pages 192–202, Berlin, Germany, July 1991. Springer.

[115] K. Jensen. Condensed state spaces for symmetrical coloured Petri nets. *Formal Methods in System Design*, 9(1/2):7–40, August 1996.

[116] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1997.

[117] Y. Jin, R. Esser, and J. W. Janneck. Describing the syntax and semantics of UML statecharts in a heterogeneous modelling environment. In M. Hegarty, B. Meyer, and N. H. Narayanan, editors, *Proceedings of the 2nd International Conference on Theory and Application of Diagrams (Diagrams'02)*, LNAI 2317, pages 320–334. Springer, April 2002.

[118] Y. Jin, R. Esser, and J. W. Janneck. Analysis-oriented semantics definition of visual languages. In *Proceedings of the IEEE Symposium on Visual Languages and Formal Methods, Human-Centric Computing (HCC'03)*, October 2003.

[119] Y. Jin, R. Esser, and J. W. Janneck. A method for describing the syntax and semantics of UML statecharts. *Software and Systems Modeling*, 2004. To appear.

[120] Y. Jin, R. Esser, and C. Lakos. Lightweight consistency analysis of dataflow process networks. In M. Oudshoorn, editor, *Proceedings of the 26th Australasian Computer Science Conference (ACSC'03)*, pages 291–300. Australian Computer Society Inc., February 2003.

[121] Y. Jin, R. Esser, C. Lakos, and J. W. Janneck. Modular analysis of dataflow process networks. In M. Pezzé, editor, *Proceedings of the 6th International Conference on Fundamental Approaches to Software Engineering (FASE'03)*, LNCS 2621, pages 184–199. Springer, April 2003.

[122] Y. Jin, C. Lakos, and R. Esser. Component-based design and analysis: A case study. In A. Cerone and P. Lindsey, editors, *Proceedings of the 1st IEEE International Conference on Software Engineering and Formal Methods (SEFM'03)*, pages 126–135. IEEE Computer Society Press, September 2003.

[123] Y. Jin, C. Lakos, and R. Esser. Modular consistency analysis of component-based designs. *Journal of Research and Practice in Information Technology*, 2004. To appear.

[124] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: Theory and applications. *Multiple-Valued Logic*, 4(1-2):9–62, 1998.

[125] M. Kaufmann and J. S. Moore. ACL2: An industrial strength theorem prover for a logic based on common Lisp. *IEEE Transaction on Software Engineering*, 23(4), 1997.

[126] J. R. Kiniry. Leading to a kind description language: Thoughts on component specification. Technical Report CS-TR-99-04, Department of Computer Science, California Institute of Technology, 1999.

[127] L. M. Kristensen and T. Mailund. A generalised sweep-line method for safety properties. In L.-H. Eriksson and P. A. Lindsay, editors, *Proceedings of the International Symposium of Formal Methods Europe (FME'02)*, LNCS 2391, pages 549–567. Springer-Verlag, July 2002.

[128] R. Kurshan. *Computer-aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[129] R. P. Kurshan. Analysis of discrete event coordinations. *LNCS 430*, pages 414–453, 1990.

[130] K. G. Larsen. *Context-Dependent Bisimulation Between Processes*. PhD thesis, University of Edinburgh, Scotland, 1986.

[131] K. G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. In M. Paterson, editor, *Proceedings of the 17th International Colloquium on Automata, Languages, and Programming (ICALP'90)*, LNCS 443, pages 526–539. Springer-Verlag, 1990.

[132] K.-K. Lau and M. Ornaghi. A formal approach to software component specification. In D. Giannakopoulou, G.T. Leavens, and M. Sitaraman, editors, *Proceedings of the Specification and Verification of Component-based Systems Workshop at OOPSLA'01*, pages 88–96, Tampa, USA, 2001.

[133] R. S. Lazic. *A Semantic Study of Data Independence with Applications to Model Checking*. PhD thesis, Oxford University Computing Laboratory, 1999.

[134] E. A. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56. Academic Press, 2002.

[135] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[136] N. G. Leveson. *SAFEWARE: System Safety and Computers*. Addison-Wesley Publishing Company, 1995.

[137] N. G. Leveson, M. P. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[138] C. Lewerentz and T. Lindner, editors. *Formal development of reactive systems: case study production cell*, LNCS 891. Springer-Verlag, 1995.

[139] J. Lilius and I. P. Paltor. vUML: A tool for verifying UML models. In *14th IEEE International Conference on Automated Software Engineering*, pages 255–258. IEEE Computer Society Press, 1999.

[140] J. Lilius and I. P. Paltor. The production cell: An exercise in formal verification of a UML model. In *Proceedings of the Hawaii International Conference on System Sciences*, 2000.

[141] X. Liu, J. Liu, J. Eker, and E. A. Lee. Heterogeneous modeling and design of control systems. In T. Samad and G. Balas, editors, *Software-Enabled Control: Information Technology for Dynamical Systems*. Wiley-IEEE Press, April 2003.

[142] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design: An International Journal*, 6(1):11–44, January 1995.

[143] D. E. Long. *Model checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie Mellon University, 1993.

[144] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CS, 1996.

[145] M. Mäkelä. Maria: Modular reachability analyser for algebraic system nets. In J. Esparza and C. Lakos, editors, *Proceedings of Applications and Theory of Petri Nets*, LNCS 2360, pages 434–444, Adelaide, Australia, June 2002. Springer-Verlag.

[146] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[147] S. McConnell. *Software Project Survival Guide*. Microsoft Press, 1998.

[148] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[149] K. L. McMillan. A compositional rule for hardware design refinement. In *[80]*, pages 24–35, 1997.

[150] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *[101]*, pages 110–121, 1998.

[151] K. L. McMillan. Circular compositional reasoning about liveness. In L. Pierre and T. Kropf, editors, *Proceedings of the 10th IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '99)*, LNCS 1703, pages 342–345, 1999.

[152] B. Meyer. Component and object technology: On to components. *Computer*, 32(1):139–140, January 1999.

[153] Microsoft Corporation. *The Component Object Model Specification*, version 0.9 edition, October 1995.

[154] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.

[155] P. J. Mosterman and H. Vangheluwe. Guest editorial: Special issue on computer automated multi-paradigm modeling. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 12(4):249–255, 2002.

[156] J. Niu, J. M. Atlee, and N. A. Day. Composable semantics for model-based notations. In W. G. Griswold, editor, *Proceedings of the 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-02)*, Software Engineering Notes 27(6), pages 149–158. ACM Press, November 2002.

[157] B. Nuseibeh, S. Easterbrook, and A. Russo. Making inconsistency respectable in software development. *Journal of Systems and Software*, 58(2):171–180, September 2001.

[158] Object Management Group. *Common Object Request Broker Architecture (CORBA/IIOP) Specification*, formal/2002-12-06 edition, 2002.

[159] Object Management Group. *OMG Unified Modeling Language Specification*, March 2003. Version 1.5, http://www.omg.org.

[160] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *[9]*, pages 411–414, 1996.

[161] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. *LNCS 607*, pages 748–752, 1992.

[162] R. F. Paige. Heterogeneous notations for pure formal method integration. *Formal Aspects of Computing*, 10(3):233–242, June 1998.

[163] A. N. Parashkevov. *Advances in space and time efficient model checking of finite state systems*. PhD thesis, School of Computer Science, University of Adelaide, 2002.

[164] A. N. Parashkevov and J. Yantchev. State efficient reachability analysis through use of pseudo-root states. In E. Brinksma, editor, *Proceedings of the 3rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, LNCS 1217, pages 50–64. Springer-Verlag, 1997.

[165] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *[57]*, pages 377–390, 1994.

[166] M. Pezzè and M. Young. Generation of multi-formalism state-space analysis tools. In S. J. Ziel, editor, *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA'96)*, pages 172–179, 1996.

[167] M. Pezzè and M. Young. Constructing multi-formalism state-space analysis tools: Using rules to specify dynamic semantics of models. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 239–250. ACM Press, 1997.

[168] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[169] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[170] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):45–60, 1981.

[171] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Model of Concurrent Systems*, volume 13 of *NATO ASI*, pages 123–144. Springer-Verlag, 1984.

[172] J. Putman and D. Hybertson. Interaction framework for interoperability and behavioral analysis. In *Proceedings of the Workshop on Object Interoperability, ECOOP'00*, 2000.

[173] S. K. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*, LNCS 2404, Copenhagen, Denmark, July 2002. Springer.

[174] A. P. Ravn and J. Staunstrup. Interface models. In *Proceedings of Codes / CASHE'94*, pages 157–164. IEEE Computer Society Press, September 1994.

[175] J. N. Reed and J. Sinclair. Combining independent specifications. In H. Hußmann, editor, *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering (FASE'01)*, LNCS 2029, pages 45–59. Springer, 2001.

[176] W. Reisig. Petri nets: An introduction. In W. Brauer, G. Rozenberg, and A. Salomaa, editors, *EATCS Monographs on Theoretical Compute Science*, volume 4. Springer-Verlag, Berlin, Germany, 1985.

[177] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall International Series in Computer Science, 1997.

[178] J. Rushby. From refutation to verification. In *Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE XIII / PSTV XX)*, October 2000.

[179] H. Saïdi. Model checking guided abstraction and analysis. In *Proceedings of the 7th International Static Analysis Symposium (SAS'00)*, LNCS 1824, 2000.

[180] H. Saïdi and N. Shankar. Abstract and model check while you prove. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, Trento, Italy, July 1999. Springer-Verlag.

[181] G. Salaün, M. Allemand, and C. Attiogbé. Foundations for a combination of heterogeneous specification components. In A. Brogi and E. Pimentel, editors, *Proceedings of the Formal Methods and Component Interaction*, ENTCS 66(4). Elsevier, 2002.

[182] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object Oriented Modeling*. John Wiley & Sons, 1994.

[183] D. B. Skillcorn. Stream languages and data-flow. In J. L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*. Prentice-Hall, 1991.

[184] I. Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

[185] G. Spanoudakis, A. Finkelstein, and D. Till. Overlaps in requirements engineering. *Automated Software Engineering*, 6(2):171–198, April 1999.

[186] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Sciences. Prentice-Hall, London, second edition, 1992.

[187] J. A. Stafford and A. L. Wolf. Annotating components to support component-based static analyses of software systems. In *Grace Hopper Celebration of Women in Computing*, Hyannis, Massachusetts, 2000.

[188] E. W. Stark. A proof technique for rely-guarantee properties. In *Proceedings of the 5th conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS 206, pages 369–391. Springer-Verlag, 1985.

[189] R. E. K. Stirewalt and L. K. Dillon. A component-based approach to building formal analysis tools. In *[5]*, pages 167–176, 2001.

[190] Sun Microsystems. *JavaBeans 1.01 Specification*, July 1997.

[191] T. Sunetnanta and A. Finkelsteing. Automated consistency checking for multiperspective software specifications. In *Proceedings of the workshop on Advanced Separation of Concerns (ICSE'01)*, 2001.

[192] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[193] K.-C. Tai and P. V. Koppol. An incremental approach to reachability analysis of distributed programs. In J. C. Wileden, editor, *Proceedings of the 7th International Workshop on Software Specification and Design*, pages 141–151. IEEE Computer Society Press, December 1993.

[194] S. Uchitel and D. Yankelevich. Enhancing architectural mismatch detection with assumptions. In *Proceedings of the Engineering of Computer Based Systems (ECBS'00)*, Scotland, UK, April 2000.

[195] A. Valmari. A stubborn attack on state explosion. In *[40]*, pages 156–165, 1990.

[196] A. Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I : Basic Models*, LNCS 1491, pages 429–528. Springer, 1998.

[197] G. Winskel. On the compositional checking of validity. In J. C. M. Baeten and J. W. Klop, editors, *Theories of Concurrency: Unification and Extension (CONCUR'90)*, LNCS 458, pages 481–501. Springer-Verlag, 1990.

[198] Y. Xiong. *An Extensible Type System for Component-Based Design*. PhD thesis, Electronics Research Laboratory, University of California, Berkeley, May 2002.

[199] B. Yang. *Optimizing Model Checking Based on BDD Characterization*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1999.

[200] W. J. Yeh. *Controlling state explosion in reachability analysis*. PhD Thesis SERC-TR-147-P, Software Engineering Research Center (SERC) Laboratory, Purdue University, 1993.

[201] D. M. Yellin and R. E. Storm. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997.

[202] P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411, October 1993.

[203] P. Zave and M. Jackson. Where do operations come from?: A multiparadigm specification technique. *IEEE Transactions on Software Engineering*, 22(7):508–528, July 1996.