The Dissertation Committee for Fei Xie

certifies that this is the approved version of the following dissertation:

Integration of Model Checking into Software Development Processes

Committee:

James C. Browne, Supervisor

E. Allen Emerson

Robert P. Kurshan

Aloysius K. Mok

Dewayne E. Perry

Integration of Model Checking into Software Development Processes

by

Fei Xie, M.S., B.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2004

To my parents, Shengguang and Qingying

to my brothers, Hang and Xiang

to my wife, Huaiyu

Acknowledgments

During my dissertation research, many people have offered their generous help. Without their help, I could never finish this dissertation. I am sincerely grateful to all of them.

My advisor, Prof. James C. Browne, has been the most important person to my dissertation research. He has offered me all the guidance and help that I could ever ask for and more. Although he leads a very busy life, he has always reserved three hours per week for our research meeting. He has always been patient listening to all the wild ideas that I came up with. His broad knowledge has guided my research through many challenges and difficulties. When I was frustrated with my research, he has always encouraged me. He has not only helped me become a better researcher, but also made me a better person. He has taught me the courtesy, integrity, and responsibility in research and other aspects of life.

Dr. Robert P. Kurshan hosted me during my three summer internships at Bell Labs. These internships laid down the foundation of my dissertation research. Bob has always been very generous with his time, knowledge, and ideas. I could stop by his office at any time to ask questions and he has always been patient and given me his best answers. When I was back to school, he has always helped me with

my research through email and phone. Whenever I asked him a research question, I have always gotten his prompt reply.

My dissertation committee members, Prof. James C. Browne, Prof. E. Allen Emerson, Dr. Robert P. Kurshan, Prof. Aloysius K. Mok, and Prof. Dewayne E. Perry, have made invaluable contributions to my dissertation. They offered their perspectives to my research, gave feedbacks to my papers, and listened to and commented on my rehearsal talks.

Prof. Stephen W. Keckler and Prof. C. Greg Plaxton were my mentors in the first year of my Ph.D. study. Steve gave me an initial education on what computer science research is about. Greg taught me the essentials for writing research papers. They continued to offer me advice and help throughout my Ph.D. study.

Dr. Vladimir Levin worked with me during my three summers at Bell Labs and one summer at Microsoft. Since my first internship at Bell Labs, we have been working together remotely. He has been very generous with his ideas and knowledge. His answers to my questions have always been insightful and comprehensive.

My life as a graduate student has been made easy by the staff members of the Department of Computer Sciences. In particular, Nancy Macmahon, the secretary to Prof. Browne, has spent a lot of time proof-reading my papers, arranging my conference travels, and most importantly making sure that I have proper financial support. Gloria Ramirez and Katherine Utz, the graduate coordinators, have walked me through the whole process of Ph.D. study, from admission to graduation. They have made all the logistics so easy.

The six years of my Ph.D. Study has been enriched by the interactions with my fellow graduate students, especially, the fellow advisees of Prof. Browne: Kevin

Kane, Nasim Mahmood, and Natasha Sharygina. Although our research topics were quite different, they have always been willing to help whenever possible. I have benefited significantly from interactions with this outstanding group of people.

This dissertation is not possible without the love, support, and encouragement from my parents Shengguang Xie and Qingying Wang, my brothers Hang Xie and Xiang Xie, and last, certainly, not least, my lovely wife Huaiyu Liu. Huaiyu is the source of my happiness, strength, and energy. She has always been there for me with her love, care, support, and encouragement while she was also working on her own Ph.D. dissertation diligently.

Fei Xie

The University of Texas at Austin

August 2004

Integration of Model Checking into Software Development Processes

Publication No. _____

Fei Xie, Ph.D.

The University of Texas at Austin, 2004

Supervisor: James C. Browne

Testing has been the dominant method for validation of software systems. As software systems become complex, conventional testing methods have become inadequate. Model checking is a powerful formal verification method. It supports systematic exploration of all states or execution paths of the system being verified. There are two major challenges in practical and scalable application of model checking to software systems: (1) the applicability of model checking to software systems and (2) the intrinsic complexity of model checking.

In this dissertation, we have developed a comprehensive approach to integration of model checking into two emerging software development processes: Model-Driven Development (MDD) and Component-Based Development (CBD), and a combination of MDD and CBD. This approach addresses the two major challenges under the following framework: (1) bridging applicability gaps through automatic translation of software representations to directly model-checkable formal represen-

tations, (2) seamless integration of state space reduction algorithms in the translation through static analysis, and (3) scaling model checking capability and achieving state space reduction by systematically exploring compositional structures of software systems. We have integrated model checking into MDD by applying mature model checking techniques to industrial design-level software representations through *automatic translation* of these representations to the input formal representations of model checkers. We have developed a *translation-based* approach to compositional reasoning of software systems, which simplifies the proof, implementation, and application of compositional reasoning rules at the software system level by reusing the proof and implementation of existing compositional reasoning rules for directly model-checkable formal representations. We have developed an *integrated state space reduction framework* which systematically conducts a *top-down* decomposition of a large and complex software system into directly model-checkable components by exploring domain-specific knowledge. We have designed, implemented, and applied a *bottom-up* approach to model checking of component-based software systems, which composes verified systems from verified components and integrates model checking into CBD. We have further scaled model checking of component-based systems by exploring the synergy between MDD and CBD, i.e., specifying components in executable design languages, and realizing the bottom-up approach based on model checking of software designs through translation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Statement

Software is pervasive, from power plants to aircrafts and to automobiles. Our everyday life depends on these artifacts. Therefore, we depend on the software systems inside these artifacts. Because of this dependency, software systems must be safe, secure, and reliable. To improve the safety, security, and reliability of software systems, these systems must be validated. The most commonly used method for software validation in the software industry is testing. However, conventional testing methods have been overwhelmed by the increasing complexity of software systems. Conventional testing methods suffer from the following problems:

- **Test case coverage.** For a complex software system, it is very difficult, if not impossible, to generate all the necessary test cases to cover all possible states or execution paths in the system.

- **Insufficient automation support.** There is insufficient tool support for automation of traditional testing methods. For instance, testing environments and test cases for software systems are still primarily generated manually.

- **Difficulties with concurrency.** Software systems are becoming increasingly concurrent, which makes testing even harder. For instance, reproducing a specific execution path of a complex concurrent software system is often difficult.

Therefore, we need advanced validation techniques for software systems. Model checking [15, 57] is one of such techniques. Model checking is a formal verification method. It is able to determine the validity of a temporal property for all possible states or execution paths in a software system to which it is applicable, given enough memory and execution time. It enjoys substantial automation support. It has been quite successful for hardware verification.

Model checking requires a model of a software system, which can be the design model, the source code, or other executable specifications of the system. It also requires a temporal property to be checked on the system. A sample temporal property is that a buffer, BUF, in the system should never overflow. (To be input by a model checker, this property must be specified in a formal property specification language such as a temporal logic.) A model checker inputs the model and the property, and conducts an exhaustive and intelligent search over the state space of the system to check whether the property holds in every state or execution path of the system. If so, the model checker reports that the property holds; otherwise, it reports a state or execution path in which the property is violated.

There are two major challenges in practical and scalable application of model checking to software systems. The first challenge is the *applicability of model check-*

ing. The input formal representations of model checkers and the widely used software representations are often significantly different. In addition, software systems often have infinite state spaces while model checkers are often restricted to finite state systems. The second challenge is the *intrinsic complexity of model checking*. The number of possible states and execution paths in a real-world software system can be extremely large, which makes naive application of model checking to such a system intractable and requires state space reduction.

The goal of this dissertation research is to seamlessly integrate model checking into two emerging software development processes: Model-Driven Development (MDD) [47], Component-Based Development (CBD) [61], and a combination of MDD and CBD. The central problem to be addressed in this research is how to overcome the two major challenges, namely, the applicability and intrinsic complexity of model checking.

1.2 Contributions and Impacts of This Dissertation

We have developed a comprehensive approach to integration of model checking into MDD, CBD, and their combination. This approach addresses the two major challenges under the following framework: (1) bridging applicability gaps through automatic translation of software representations to directly model-checkable formal representations, (2) seamless integration of state space reduction algorithms in the translation through static analysis, and (3) scaling model checking capability and achieving state space reduction by systematically exploring compositional structures of software systems. We have integrated model checking into MDD by applying mature model checking techniques to industrial design-level software representations

through *automatic translation* of these representations to the input formal representations of model checkers [68, 69, 70]. In the translation, we have applied many state space reduction algorithms to software representations under an *integrated model and property translation framework* [70] in which the translation of a model depends on the property to be checked and the state space reduction algorithms to be applied. We have developed a *translation-based* approach [67] to compositional reasoning [53, 1, 3, 46, 4, 19] of software systems, which simplifies the proof, implementation, and application of compositional reasoning rules at the software system level by reusing the proof and implementation of existing compositional reasoning rules for directly model-checkable formal representations. We have developed an *integrated state space reduction framework* [65], which systematically conducts a *top-down* decomposition of a large and complex software system into directly model-checkable pieces by exploring domain-specific knowledge. We have designed, implemented, and applied a *bottom-up* approach [66] to model checking of component-based software systems, which composes verified systems from verified components and integrates model checking into CBD. We have further scaled model checking of component-based systems by exploring the synergy between MDD and CBD, i.e., specifying components in executable design languages and realizing the bottom-up approach based on model checking software designs through translation.

1.2.1 Translating Software Designs for Model Checking

MDD is becoming widely used, especially in the domain of embedded systems, for instance, avionics flight control systems. In MDD, executable design-level models of a software system are constructed and implementations of the system are com-

piled from these models based on predefined templates. Executable design-level models are the foundation of MDD, and they also contribute to overcoming the two major challenges in software model checking. These models are executable, therefore, are amenable to automatic translation for model checking. These models are on design level, therefore, are more abstract and have smaller state spaces than implementation-level representations of software systems.

We have developed the *ObjectCheck* toolkit [69] which provides fully automatic model checking support for Executable UML (xUML) [47], an executable design-level software specification language, based on *translation*. A design model in xUML and a property to be checked are translated into S/R [28], the input formal language of the COSPAN [28] model checker, and the resulting S/R model and property are model checked by COSPAN. ObjectCheck has been applied to design models of real-world software systems: a NASA robot controller [36], a distributed transaction system [64], and the UC Berkeley TinyOS runtime system for networked sensors [30]. In the case study on the NASA robot controller, ObjectCheck successfully checked 22 properties on the robot controller, including both safety and liveness properties, and revealed 6 serious logical errors that had not been detected by conventional testing [60].

In developing the ObjectCheck toolkit, we have designed and developed an integrated model and property translation framework, which addresses major issues in translator construction: translator architecture, property specification and translation, semantics mapping from software languages to formal languages, and transformations for state space reduction. The framework was motivated by the observation that model checking of a property on a software system *only* requires that

the behaviors of the system, which are related to the property, be preserved in the resulting formal model. Under this framework, the artifact to be translated consists of a model of a software system and a property to be checked. Many state space reduction algorithms, such as static partial order reduction [39] and transition compression [40], are applied in the translation to reduce the state space of the model with respect to the property. This framework provides a unified view for application of various state space reduction algorithms in translation. These algorithms basically project the software system to a formal model with a minimal state space, with respect to the property. This framework may benefit the development of the translators from other software specification languages to directly model-checkable formal languages.

1.2.2 Translation-Based Compositional Reasoning

One of the most powerful state space reduction algorithms is compositional reasoning where model checking a property on a system is accomplished by decomposing the system into components, checking component properties locally on the components, and deriving the property of the system from the component properties. Application of compositional reasoning to software systems requires establishing a compositional reasoning rule in the semantics of these systems, proving the correctness of the rule, and implementing the rule. Directly proving the correctness of compositional reasoning rules for software systems is often difficult since software specification languages are complex in syntax and semantics and these languages often have varying operational semantics.

We have developed Translation-Based Compositional Reasoning (TBCR) [67],

an approach to application of compositional reasoning in the context of model checking software systems through translation. In this approach, given a translation from a software semantics to a directly model-checkable formal semantics, a compositional reasoning rule is established in the software semantics and mapped to an equivalent rule in the formal semantics based on the translation. The correctness proof of the compositional reasoning rule in the software semantics is established based on this mapping and the correctness proof of the equivalent rule in the formal semantics. The compositional reasoning rule in the software semantics is implemented and applied based on the translation from the software semantics to the formal semantics and through reusing the implementation of the equivalent rule in the formal semantics. We have realized TBCR for a commonly used software semantics, the Asynchronous Interleaving Message-passing (AIM) semantics, and applied it in the integrated state space reduction framework and in composition of verified systems from verified components. Proof, implementation, and application of compositional reasoning rules for the AIM semantics were found to be greatly simplified.

1.2.3 Integrated State Space Reduction Framework

We have developed an integrated state space reduction framework [65] for model checking executable software system designs. The essence of this framework is *systematic decomposition and reduction* of a large and complex software system into small and directly model-checkable pieces in a *top-down* fashion. Model checking of a complex software system typically requires application of many state space reduction algorithms and these algorithms often interact with one another. Under this framework, state space reduction algorithms, such as compositional reasoning,

abstraction [17], symmetry reduction [16], and partial order reduction [25, 52, 62], are applied in a systematic and integrated way to an executable software system design in xUML before and during its translation and to the resulting formal model. Interactions among these algorithms are explored to maximize the aggregate effect of state space reduction. Although presented in the context of model checking executable software system designs through translation, the framework can be readily adapted to model checking of other software representations through translation.

State space reduction algorithms such as compositional reasoning, abstraction, and symmetry reduction, whose effectiveness depends on structures and behaviors of software systems, can be readily formulated on design models due to the fact that execution behaviors of different components are more visible at the design level. Many software system designs are constructed following domain-specific design patterns that provide information about structures and behaviors of these systems. These facts suggest instantiating the integrated state space reduction framework for different application domains based on domain-specific design patterns. We have instantiated the framework for distributed transaction systems and applied the instantiation to the design model of an online ticket sale system [64]. With this framework, we were able to greatly extend the scale of model-checkable software system designs. For instance, in checking the online ticket sale system, a verification task (checking a property on the system) which originally required 152.79 megabytes and 16273.7 seconds to complete was reduced to 7 verification subtasks (checking sub-properties on components of the system), each of which required at most 0.95 megabytes and 1.82 seconds.

1.2.4 Verified Systems by Composition from Verified Components

CBD, developing software systems through composition of components and reuse of components, is one of the most important technical initiatives in software engineering. CBD and model checking are synergistic. CBD introduces compositional structures, clean component interfaces, and standard composition rules to the systems being built, which may reduce the state spaces that model checkers have to handle. Model checking is particularly effective at detecting coordination errors which frequently result from component compositions and are notoriously difficult to detect. The process of CBD provides a natural means of combining compositional reasoning and abstraction algorithms to reduce the complexities of the state spaces that must be explored directly by model checkers.

We have developed an approach [66] to integration of model checking into CBD. This approach assists in development of safe, secure, and reliable component-based software systems and reduces the complexity of verifying these systems by utilizing their compositional structures. Temporal properties of a software component are specified, verified, and packaged with the component. Selection of a component for reuse considers not only its functionality but also its temporal properties. When a component is composed from other components, a property of the component is model-checked on an abstraction of the component. The abstraction is constructed from environment assumptions of the component and verified properties of its sub-components. The essence of this approach is establishment and reuse of temporal properties of components in the bottom-up composition of components.

We have further scaled model checking of component-based systems by exploring the synergy between MDD and CBD and realizing the bottom-up approach

based on the ObjectCheck toolkit. Components are specified in executable design languages. Properties of primitive components (components that are not composed from other components) are model-checked on the executable designs of the components with the ObjectCheck toolkit.

This approach has been applied to improve reliability of instances (installations that can execute on sensor hardware) of TinyOS, a component-based run-time system for networked sensors. A coordination bug in TinyOS was detected and our approach has led to an order-of-magnitude reduction on the complexity of model checking TinyOS components.

1.3 Dissertation Outline

The reminder of this dissertation is organized as follows. In Chapter 2, we present background material on MDD, CBD, and model checking. In Chapter 3, we discuss model checking software designs through translation. In Chapter 4, we present translation-based composition reasoning. In Chapter 5, we present the integrated state space reduction framework. In Chapter 6, we discuss composition of verified systems from verified components. In Chapter 7, we conclude this dissertation and discuss future research directions.

Chapter 2

Background

In this chapter, we first introduce the two software development processes: Model-Driven Development and Component-Based Development, and their combination. We then briefly discuss model checking, the COSPAN [28] model checker that we employ in our research, and the automata-theoretic approach [37] to model checking implemented by COSPAN.

2.1 Model-Driven Development (MDD)

MDD is an emerging approach to the development of software systems. The Model-Driven Architecture (MDA) [51] proposed by Object Management Group (OMG) provides a framework for MDD. The goal of MDA is to separate business or application logics from underlying implementation platform technologies. MDD achieves this goal by building Platform-Independent Models (PIMs) of applications. These applications can be realized on many different platforms by implementing Platform-Dependent Models (PDMs) of these applications from their PIMs.

There are many languages for specification of PIMs, for instance, various dialects of the Unified Modeling Language (UML) [50]. These PIM languages can be categorized as follows: non-executable and executable. A PIM language is executable if its syntax and semantics enable a PIM specified in this language to be simulated in an execution simulator or to be compiled to source codes in conventional program languages or to executables. Both non-executable PIM languages and executable PIM languages have their own advantages. For instance, non-executable PIM languages may offer higher levels of abstraction and may be more expressive. Executable PIM languages enable early validation and verification of a software system being developed since they have complete execution semantics. Early error detections are critical to reducing development and maintenance costs of software systems. MDD using executable PIM languages are becoming increasingly popular for the embedded system domain and the web-based system domain. The general workflow of a MDD process using an executable PIM language is shown in Figure 2.1. This MDD process has the following steps:

1. **Analysis and Design.** Based on the requirements of the system being developed, the system designers conduct analysis and design. The work product of this step is an executable platform-independent design model of the system.

2. **Design Validation.** Since the design model built in Step 1 is executable, it can then be validated for correctness. Conventional validation methods such as testing through simulated execution are supported by commercial integrated development environments for executable PIM languages, for instance, Bridgepoint [56] and iUML [9]. These validation methods are often insufficient due to problems such as limited test case coverage: it is often difficult, if not im-

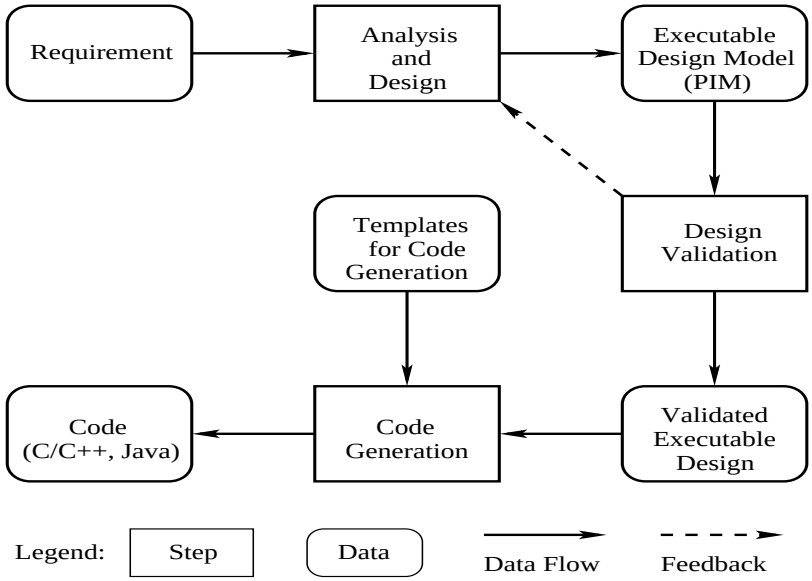

Figure 2.1: A MDD process using an executable PIM specification language

possible, to generate all the necessary test cases to cover all possible states or execution paths in a complex design model.

3. **Code Generation.** Once the design model is sufficiently validated, code in conventional program languages such as C/C++ and Java can be generated from the validated design model. Code generation is based on a set of predefined templates which specify how semantic entities in the design model can be compiled into code.

An additional step of the MDD process, which is not shown in Figure 2.1, is the design, implementation, and validation of the templates for code generation. A new set of templates need to be designed, implemented, and validated when a new platform technology is selected. If a single system is to be developed on this platform,

then this step could be expensive. However, if a family of systems are to be developed on this platform, the cost for design and validation of these templates can be amortized over these systems.

To integrate model checking into MDD, it is natural to select an executable PIM language since model checking requires that the model to be checked have complete execution semantics. The executable PIM language that we select is executable UML (xUML) [47]. In xUML, a system is composed of instances of classes which are either active, having dynamic behaviors, or passive, having no dynamic behaviors and used to store data. There can be association and generalization relationships among classes. A large system can be recursively partitioned into packages, which are groups of classes closely coupled by associations and generalizations. Under a generalization, subclasses inherit attributes and message types (if defined) of the superclass, but the subclasses do not inherit the state model of the superclass. Every active subclass defines its own state model.

Behaviors of instances of an active class are specified by a state model that consists of: states, state transitions, actions, and message types. Each state transition is labeled by a message type and messages of the type enable the transition. An action is an operation that is associated with a state and must be executed when an instance arrives in that state. Actions can be divided into five categories: (1) computation actions; (2) read or write actions that read or write attributes of class instances, or create or delete class instances; (3) messaging actions that send messages to active class instances; (4) composite actions that are control structures and recursive structures that permit complex actions to be composed from simpler actions; (5) collection actions that apply other actions to collections of elements.

xUML has an asynchronous interleaving message-passing semantics with the following properties:

- **Creation and deletion of class instances.** Class instances can be created at system initialization and can also be dynamically created or deleted during system execution.
- **Asynchronous message passing.** Active class instances communicate via asynchronous message passing. Every active class instance has a private message queue that is FIFO and infinite.
- **Active class instance scheduling.** At any given point of a system execution, exactly one active class instance is nondeterministically scheduled to execute from among all active class instances that are ready. The scheduled active class instance either performs a state transition by consuming the first message in its private message queue, or executes the action associated with its current state. The execution of an action or a state transition is run-to-completion.
- **Disposition of unexpected messages.** When a class instance notices the first message in its message queue is unexpected in its current state, it can choose to ignore the message or to flag a system error. Message disposition rules are attached to states.

2.2 Component-Based Development (CBD)

In CBD [61], software systems are developed through composition of components. A component is a software artifact that has a clearly defined functionality and a well

defined interface and can be reused across software systems. CBD has the following appealing promises:

- **Better structures for software systems**. CBD introduces compositional structures to software systems being developed, provides well defined interfaces for components, and specifies composition rules for components explicitly. These structures enable rapid construction of software systems of high quality at a low cost.

- **Reuse of development efforts.** CBD supports component reuse across software systems, which helps reuse previous development efforts and share accumulated knowledge. It also helps amortize the cost of developing components.

- **Easy maintenance and evolution of software systems.** CBD makes software systems easy to maintain and to evolve. Since a component has a well defined interface, if the component need to be maintained or evolved, it can be replaced with a new one without changing other components connected to it in a software system.

Despite the great promises of CBD, its adoption in the software industry has been slow. There are several major impediments to adoption of CBD. First, the state-of-art practices of software development are not rigorous enough. Software engineers tend to start coding before they have done a thorough design of the software systems or components being developed. The interfaces among components are often poorly designed and frequently modified, which makes reuse of components difficult. CBD is effective only if the set of components for a family

of software systems are carefully designed together. Secondly, hardware systems of high performance were quite expensive. To develop a system that meets performance requirements at a reasonable price, many software optimizations had to be applied. These optimizations often blur component boundaries, change component interfaces, and make components application-specific and difficult to reuse. The situation has improved dramatically. As fast CPUs and large memory modules become available at increasingly low prices, in many application domains, speeding up the software development cycle, reusing previous development efforts, and improving software quality have become more important than optimizing software for performance. The demands for highly safe, secure, and reliable software systems motivate increasing adoption of rigorous design and development procedures. These factors encourage adoption of CBD in many application domains such as embedded systems and web-based systems. A common characteristic of these domains is that a family of applications often reuse a common set of components and each application typically has only a few application-specific components.

In the embedded system domain, a family of applications are often built upon a specific hardware platform. Design and implementation of applications on such a platform requires domain-specific expertise. Such expertise may be scarce, which makes reuse critical. For instance, there are increasing demands for networked sensor systems. It is desirable for engineers in various fields to design and implement their own sensor systems in a timely fashion without obtaining the full knowledge of how to program on a specific sensor hardware. This has led to adoption of CBD in networked sensor system development, for instance, the TinyOS [30] project from the University of California, Berkeley.

In the web-based system domain, large-scale distributed systems are developed by reusing existing web services, aggregating web services to build larger web service, and developing new services as necessary. Web-based systems are the foundation for enterprise information and decision support systems. The business logics of enterprises are ever-changing. New systems must be developed in a timely fashion and at a low cost, which makes CBD ideal for developing such systems. In this domain, components are web services. A web service has a well-defined interface often specified in the Web Service Definition Language (WSDL) [63]. A basic web service often provides a fundamental functionality such as database query processing. An aggregated web service often implements a business logic such as processing a transaction.

2.3 Combination of MDD and CBD

There is an emerging trend to combine CBD with MDD to further simplify software development and raise the abstraction level on which software systems are being developed. For instance, several design-level executable specification languages, such as Business Process Definition Language for Web Services (BPEL4WS) [34], have been proposed for specification of web services. These languages can be used to build a web service from scratch and can also be used to aggregate existing web services to build a larger web service. Web services specified in these languages can be directly compiled to deployable executables.

2.4 Model Checking

Model checking [15, 57, 17] is an automatic technique for verifying finite state concurrent systems. It has been successfully applied to verification of both hardware and software systems, for instance, complex circuit designs and communications protocols. Application of model checking to a hardware or software system consists of the following tasks:

- **Model acquisition.** A model of the system must be obtained either through manual construction or automatic generation. Automatic generation has been the main stream since manual construction is often laborious and error-prone. In some cases, automatic generation is mainly a compilation task. However, in many other cases due to time and memory limitations, automatic generation often involves application of state space reduction algorithms to remove unnecessary details from the model.

- **Property specification.** The properties to be verified on the model must be formulated. These properties are commonly specified in some logical formalism, for instance, a temporal logic, which asserts on the behaviors of systems over time. Property specification is a challenging task. Model checking is capable of verifying if the model conforms to the properties given enough time and memory. However, it is incapable of verifying whether the properties are correctly and completely specified. Properties are often specified manually. For some properties that are common and well-understood, it is possible to generate these properties automatically.

- **Verification.** Once the system is modeled and the properties are specified, a model checker can be applied to verify the properties on the model. The model checker conducts an exhaustive but intelligent search over the state space of the model to verify if the properties hold in every possible state or execution path of the model. If so, the model checker reports that the properties hold on the model. If not, let that the model checking process terminates, the model checker generates an error trace which can be used as a counterexample for the property and help the system developers debug the system. It is possible that the error has been introduced in model acquisition or property specification. In such a case, the model must be fixed or the property must be corrected. If the verification does not terminate due to time and memory limitation, it is often required to change the parameters of the model checker or re-construct the model using additional state space reduction algorithms.

A major advantage of model checking over other validation methods such as testing and deductive verification is that it can be performed automatically. Model checking is often restricted to finite-state systems. (There are model checkers for certain types of infinite-state systems.) This restriction enables full automation of model checking. Model checkers normally use an exhaustive search of the state space of the system to determine if a property holds on the systems. Given sufficient resources, the search will eventually terminate.

A major challenge to application of model checking is the state space explosion problem. As the number of components or variables in a system becomes large, the state space of the system can become arbitrarily large. It is very memory and time consuming to cover all the possible states or execution paths in such a

system and, in many cases, it is even impossible. There has been a large amount of research [17] on state space reduction algorithms which can reduce the sizes of state spaces that must be exhaustively searched.

There are many approaches to model checking. We choose the automata-theoretic approach [37] to model checking. In this approach, a system is modeled by an automaton P and a property to be checked is modeled by an automaton T. The verification consists of checking whether the language of P is contained in the language of T, $\mathcal{L}(P) \subset \mathcal{L}(T)$, known as the language containment test. Typically, P is not monolithic, but is represented as a synchronous parallel composition $P = P_1 \otimes \ldots \otimes P_k$ of component processes all modeled as automata. A model checker that implements this approach is COSPAN [28]. COSPAN checks language containment by either an explicit state space enumeration algorithm or a symbolic (BDD-based or SAT-based) algorithm.

COSPAN inputs the S/R automaton language. In S/R, a system is composed of synchronously interacting processes (or automata). A process consists of state variables, selection variables, inputs, state transition predicates, and selection rules. Selection variables define the outputs of the process. Each process imports a subset of all the selection variables of other processes as its inputs. State transition predicates update state variables as functions of the current state, selection variables, and inputs. Selection rules assign values to selection variables as functions of state variables. Such a function is nondeterministic if several values are possible for a selection variable in a state. The "selection/resolution" execution model of S/R is clock-driven, synchronous, and parallel, under which a system of processes behaves in a two-phase procedure every logical clock cycle:

- [1: Selection Phase] Every process "selects" a value possible in its current state for each of its selection variables. The values of the selection variables of all the processes form the global selection of the system.

- [2: Resolution Phase] Every process "resolves" the current global selection simultaneously by updating its state variables upon enabled state transition predicates.

COSPAN implements a number of state space reduction algorithms including localization reduction, symmetry reduction, and user-defined homomorphic reduction [37]. When COSPAN applies these state space reduction algorithms, it transforms a given S/R model into a semantically equivalent one with a reduced state space, with respect to a property or a set of properties. Other state space reduction algorithms such as predicate abstraction [26] and assume-guarantee style [1, 3, 46, 4] of compositional reasoning have been implemented as transformations to an S/R model before the model is model-checked by COSPAN.

Chapter 3

Model Checking Software Designs through Translation

3.1 Motivation and Overview

Approaches to software model checking can be roughly categorized as follows:

1. Manually creating a model of a software system in a directly model-checkable formal language and model checking the model in lieu of the system;

2. Subsetting a software implementation language and directly model checking programs written in this subset;

3. Subsetting a software implementation language and translating this subset to a directly model-checkable formal language;

4. Abstracting a system implemented in a software implementation language and translating the abstraction into a directly model-checkable formal language;

5. Developing a system in an executable software design specification language and translating the design into a directly model-checkable formal language;

6. Model checking a property on a system through systematic testing of the execution paths associated with the property.

Categories 3, 4, and 5 cover a large fraction of the approaches to software model checking, such as [5, 18, 29, 33, 43, 69], all of which require translation from a software language or an abstraction specification language to a directly model-checkable formal language. Translation helps avoid the "many models" problem: as a system evolves, models of the system are manually created and may contain errors or inconsistencies. Translation also enables application of state space reduction algorithms by transforming the designs, implementations, and abstractions being translated. There has, however, been little systematic consideration of issues involved in translating software specification languages used in software development to directly model-checkable formal languages.

This chapter identifies and formulates several major issues in translating executable design level software specification languages to directly model-checkable formal languages. Solutions to these issues are defined, described, and illustrated in the context of developing the translator [69] from xUML [47], an executable design level specification language, to S/R [28], the input language of the COSPAN [28] model checker. (Another translator [42], which translates SDL [35] to S/R, is also referred to as we discuss issues related to reuse of translator implementation.)

Model checking of a property on a software system through translation *only* requires that the behaviors of the system related to the property be preserved in the resulting formal model. The artifact to be translated consists of a model of a software

system and a property to be checked. This *integrated model and property translation* provides a natural framework for generating a formal model that preserves only the behaviors required for model checking a given property and has a minimal state space. Under this framework, the following issues in translation of executable design level software specification languages have been identified and formulated in developing the xUML-to-S/R translator:

- **Translator architecture.** The architecture of translators should simplify implementation and validation of translation algorithms and transformation algorithms for state space reduction, and also enable reuse of these algorithms.

- **Semantics translation from a software language to a formal language.** Model checking of software through translation requires correct semantics translation from a software specification language to its target formal language. The semantics of the source software language and the semantics of the target formal language may differ significantly, which may make the translation non-trivial.

- **Property specification and translation.** Effective model checking of software requires specification of properties on the software level and also requires integrated translation of these properties into formal languages with the system to be checked.

- **Transformations for state space reduction.** Many state space reduction algorithms can be implemented as source-to-source transformations.

- **Translator validation and evolution.** Translators must be validated for correctness. They must be able to adapt to evolution of source software lan-

guages and target formal languages, and incorporation of new state space reduction algorithms.

These issues arise generally in translation of software specification languages for model checking. We have chosen executable design level software specification languages as our representations for software systems for the following reasons:

- These languages are becoming increasingly popular in industry and development environments for these languages are commercially available.
- These languages have complete execution semantics that enable application of testing for validation and also enable application of model checking for verification.
- A design in these languages can be compiled into implementation level software specification languages and also can be translated into directly model-checkable formal languages. This establishes a mapping between the implementation of the design and the formal model of the design that is model checked, which avoids the "many models" problem.
- These languages require minimal subsetting to enable translation to directly model-checkable formal languages.

The balance of this chapter is organized as follows. In Sections 3.2, 3.3, 3.4, 3.5, and 3.6, we elaborate on these issues and discuss their solutions in the context of the xUML-to-S/R translator. We present the ObjectCheck toolkit that encapsulates the xUML-to-S/R translator in Section 3.7, briefly touch on several case studies using the xUML-to-S/R translator in Section 3.8, discuss related work in Section 3.9, and summarize in Section 3.10.

3.2 Translator Architecture

This section presents a general architecture for translators from software specification languages to directly model-checkable formal languages and briefly discusses the functionality of each component in this architecture. The emphasis is on the Common Abstraction Representation (CAR), the intermediate representation of the translation process. Many of the important functionalities of translators are implemented as source-to-source transformations on the software model to be translated or on the CAR.

3.2.1 A General Architecture for Translators

A general architecture for translators is shown in Figure 3.1. A notable feature of

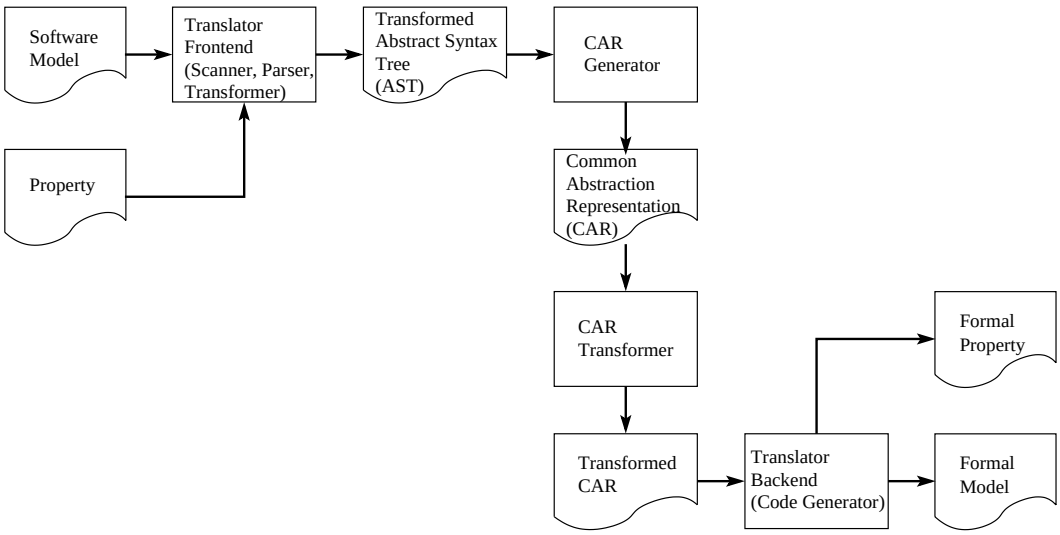

Figure 3.1: Translator architecture

this architecture is that the software model and the property to be checked on the

model are processed in an integrated fashion by each component. The frontend of the translator not only constructs the Abstract Syntax Tree (AST) of the software model, but also transforms the AST with respect to the property by applying source-to-source transformations such as the loop abstraction [59]. These transformations are partially guided by directives written in an annotation language to be discussed in Section 3.5.1. Functionalities of other components are discussed in Section 3.2.2 after we introduce the CAR.

3.2.2 Common Abstraction Representation (CAR)

A CAR is a common intermediate representation for translating several different software languages. It captures abstract concepts of the basic semantic entities of these languages and is designed to be a minimal representation of the core semantics of these languages. A CAR has been derived for the development of the xUML-to-S/R and SDL-to-S/R translations. The basic entities in this CAR include a system, a process, a process buffer, a message type, a message, a variable type, a variable, and an action. A process entity is structured as a graph whose nodes are states, conditions, and actions and whose edges are transitions. Actions are input, output, assignment, and etc.

Entities in a CAR may have parameterized definitions. Semantics of such entities can be exactly specified only by referring to a specific source language. For instance, in an xUML process, actions are associated with states while in a SDL process, actions are associated with transitions. For translation of a specific source language, a *profile* of the CAR is defined. The profile is a realization of the CAR which includes the CAR entities necessary for representing the source

language and realizes the CAR entities with parameterized definitions according to the semantics of the source language. Each model in the source language is represented by an instance of the CAR profile. The CAR profile thus inherits its semantics from the source language. This semantics is mapped to the semantics of a target language by a translator backend. CAR profiles for different source languages require different translation backends to a target language. These backends share translation procedures for a CAR entity if the entity has the same semantics in the corresponding source languages. Semantic entities of a source language that are not in the CAR are either reduced to the entities that are in the CAR or included as extensions in the CAR profile for the source language. Having a CAR and different CAR profiles for different source languages offers the following benefits:

- A CAR profile only contains the necessary semantic entities for a source language. It is easier to construct and validate the translation from the CAR profile to the target language than the direct source-to-target translation.
- The simplicity of the CAR profile simplifies the implementation and validation of transformations for state space reduction.
- The CAR enables reuse of the translation algorithms and the transformation algorithms for the semantic entities shared by different profiles of the CAR.

There is often a significant semantics gap between a source language and a target language, which makes a single-phase direct translation difficult. Having a CAR allows us to divide the translation from a source language to a target language into three phases: (1) the *CAR instance construction* phase, (2) the *CAR instance transformation* phase, and (3) the *target language code generation* phase.

In the *CAR instance construction* phase, a model in the source language is scanned, parsed, and transformed, and a CAR instance is then constructed. Complex semantic entities in the model are reduced to basic semantic entities in the CAR. For instance, in xUML, there are several different loop structures in the action language such as a *for* loop, a *while* loop, and a *do* loop. All these loop structures are reduced to a simple loop structure composed of a condition, the loop body, and *goto* actions. Implicit semantic entities are made explicit in the CAR instance. For instance, there is an implicit message buffer for each class instance in an xUML model, which is not explicitly represented in the xUML model. To be translated, such buffers are made explicit.

In the *CAR instance transformation* phase, the CAR instance is transformed by source-to-source transformations for state space reduction. CAR provides a common representation on which transformations for state space reduction such as static partial order reduction [39] can be implemented. Since the CAR profiles for different source languages share semantic entities, transformations implemented on these semantic entities may be reused in translation of different source languages.

In the *target language code generation* phase, a model in the target language is generated from the transformed CAR instance. For each semantic entity in the CAR, a code generation procedure is defined. As the AST of a CAR instance is traversed, if a semantic entity is identified, the corresponding code generation procedure is invoked to emit codes in the target language. An entity in the CAR may have different semantics when used in translation of different source languages. Therefore, the code generation procedures for translating this entity may be different for different source languages. For instance, in xUML each class instance has a

message buffer while in SDL each process has a message buffer. However, in xUML and SDL the message buffers have different semantics. In xUML, a class instance can consume, discard, and throw an exception on a message in its message buffer. In SDL, a process can save a message in its buffer and consume it in the future. The translation procedures for translating an xUML message buffer and an SDL message buffer are, therefore, different.

3.3 Semantics Translation from Software Language to Formal Language

To translate a software language for model checking, a proper target formal language is selected. After selection of the target language, a translatable subset of the software language is derived. This subset is mapped to the CAR by reducing complex semantic entities in the source language to simple semantic entities in the CAR. The simplified semantics of the source language is then simulated with the semantics of the target language. We discuss these steps in the context of the xUML-to-S/R translation.

3.3.1 Selecting Target Formal Language

There are many directly model-checkable formal languages. Promela [32], SMV [45], and S/R [28] are among the most widely used. These languages have various semantics. Their corresponding model checkers, SPIN [32], SMV [45], and COSPAN [28], support different sets of search algorithms and state space reduction algorithms. Appropriate selection of a target formal language should consider three factors: *application domain*, *semantics similarity*, and *model checker support*. We considered

these factors synergistically in selecting the target language for translating xUML.

- **Application domain.** While this chapter is concerned only with translation of a software design, the ultimate goal of this project is hardware/software co-verification. xUML is widely used in development of embedded systems which often requires hardware/software co-design and co-verification. Such a system, on different levels of abstraction, may exhibit both hardware-specific (tighter synchronization) and software-specific (looser synchronization) behaviors.

- **Semantics similarity.** The asynchronous interleaving semantics of xUML is close to the semantics of Promela, which would simplify the translation, while both SMV and S/R have synchronous parallel execution semantics.

- **Model checker support.** In practical model checking, especially in co-verification, the widest range of search algorithms and state space reduction algorithms is desired since it is not clear that any algorithm is superior for a well-identified class of systems. A system that has both software and hardware components may often benefit from symbolic search algorithms based on BDDs and SAT solvers which are not available in SPIN. SMV provides BDDs and SAT based symbolic search algorithms. However, Depth-First Search (DFS) algorithms with explicit state enumeration, which have demonstrated their effectiveness in verification of many software-intensive systems, are not available in SMV. COSPAN offers both symbolic search algorithms and DFSs with explicit state enumeration. COSPAN supports a wide range of state space reduction algorithms such as localization reduction [28], static partial order reduction [39], and a prototype implementation of predicate abstraction [49].

Based on the above, we selected S/R as the target language at the cost of a non-trivial xUML-to-S/R translation.

3.3.2 Subsetting Software Language

Software languages such as xUML may have multiple operational semantics and may also have semantic entities not directly translatable to the selected target language. For model checking purposes, a subset of the software language must be derived for a given application domain. This subset must have a clean operational semantics suitable for the application domain. Semantic entities that are not directly translatable, such as continuous data types, must be either excluded from the subset or discretized and simulated by other semantic entities. Infinite-state semantic entities may be directly translated or be bounded and then translated depending on whether the target language supports infinite-state semantic entities or not. If a target formal language permits some infinite-state semantic entities, necessary annotations may also need to be introduced for the subset so that infinite-state semantic entities in the subset can be properly translated.

In the xUML-to-S/R translation, we adopt an asynchronous interleaving semantics of xUML (see Section 3.3.4) while xUML has other semantics such as asynchronous parallel. Continuous data types such as float can be simulated by discrete data types such as integer if such a simulation does not affect the model checking result. Since S/R does not support infinite semantic entities, infinite data types and infinite message queues must be bounded implicitly by convention or explicitly by user annotations.

3.3.3 Mapping Source Software Language to CAR

After the translatable subset of the source software language is derived, a CAR profile is identified accordingly. The CAR profile only contains the basic entities necessary for representing the source language subset. A mapping is then established from the source language subset to the CAR profile. Complex semantic entities in the source language are reduced to simple semantic entities in the CAR. For instance, in xUML a state action can be a collection action that applies a sub-action to elements of a collection in sequence. The collection action is reduced into a loop action with a test checking whether there still are untouched elements in the collection, and with the sub-action as the loop body. After the mapping is established, the semantics of the CAR profile is decided by the semantics of the source language and the mapping.

3.3.4 Simulating Source Semantics with Target Semantics

The mapping from the source language to the CAR profile removes complex semantic entities from the source semantics. To complete the translation to the target language, only this simplified form of the source semantics must be simulated with the target semantics. We first sketch the semantics of xUML and S/R, then discuss how the asynchronous semantics of xUML is simulated with the synchronous semantics of S/R and how the run-to-completion requirement of xUML is simulated.

Background: semantics of xUML and S/R

xUML has an asynchronous interleaving message-passing semantics. In xUML, a system consists of a set of class instances. Class instances communicate via asyn-

chronous message-passing. The behavior of each class instance is specified by an extended Moore state model in which each state may be associated with a state action. A state action is a program segment that executes upon entry to the state. In an execution of the system, at any given moment only one class instance progresses by executing a state transition or a state action in its extended Moore state model. S/R has a synchronous parallel semantics. In S/R, a system consists of a set of automata. Automata communicate synchronously by exporting variables to other automata and importing variables from other automata. The system progresses according to a logical clock. In each logical clock cycle, each automaton moves to its next state according to its current state and the values of the variables it imports.

Simulation of asynchrony with synchrony

The asynchronous interleaving execution of an xUML system is simulated by the synchronous parallel execution of its corresponding S/R system as follows. Each class instance in the xUML system is mapped an automaton in the S/R system. An additional automaton, *scheduler*, is introduced in the S/R system. The *scheduler* exports a variable, *selection*, which is imported by each S/R automaton corresponding to an xUML class instance. At any given moment, the *scheduler* selects one of such automata through setting *selection* to a particular value. Only the selected automaton executes a state transition corresponding to a state transition or a state action in the corresponding xUML class instance. Other automata follow a self-loop state transition back to their current states.

The asynchronous message-passing of xUML is simulated by synchronous variable-sharing of S/R through modeling the message queue of a class instance as a

separate S/R automaton. Let automata IP_1 and IP_2 model two class instances and automata QP_1 and QP_2 model their corresponding private message queues. The asynchronous passing of a message, m, from IP_1 to IP_2 is simulated as follows: [1: $IP_1 \rightarrow QP_2$] IP_1 passes m to QP_2 through synchronous communication; [2: Buffered] QP_2 keeps m until IP_2 is ready for consuming a message and m is the first message in the queue modeled by QP_2. [3: $QP_2 \rightarrow IP_2$] QP_2 passes m to IP_2 through synchronous communication.

Simulation of run-to-completion execution

A semantic requirement of xUML is the run-to-completion execution of state actions, i.e., the executable statements in a state action must be executed consecutively without being interleaved with state transitions or executable statements from other state actions. This run-to-completion requirement is simulated as follows. An additional variable, *in-action*, is added to each S/R automaton corresponding to an xUML class instance. All *in-action* variables are imported by the *scheduler*. When an automaton is scheduled to execute the first statement in a state action, it sets its *in-action* to true. When the automaton has completed with the last statement in the state action, it sets its *in-action* to false. The scheduler continuously schedules the automaton until its *in-action* is set to false.

3.4 Property Specification and Translation

Since the entire translation process is property-dependent, properties must be specified at the level of and in the name space of software systems. Additionally, software level property specification enables software engineers who are not experts in model

checking to formulate properties. We discuss software level property specification and translation of software level properties in terms of xUML and a linear-time property specification language, but the arguments carry over for other software specifications and temporal logics. Two issues related to property specification and translation: (1) automatic generation of properties and (2) translation support for compositional reasoning, conclude this section.

3.4.1 Software Level Property Specification

An xUML level property specification language, which is linear-time and with the expressiveness of ω-automata, has been defined. This language consists of a set of property templates that have intuitive meanings and also rigorous mappings into the FormalCheck property specification language [38] which is written in S/R. The templates define parameterized automata. Additional templates can be formulated in terms of the given ones, if doing so simplifies the property specification process. A property formulated in this language consists of declarations of propositional logic predicates over semantic entities of an xUML model and declarations of temporal predicates. A temporal predicate is declared by instantiating a property specification template: each argument of the template is replaced by a propositional logic expression composed from previously declared propositional predicates.

To further simplify property specification, for an application domain, frequently used property templates and customized property templates are included in a domain-specific property template library based on previous verification studies in the domain. These property templates are associated with domain-specific knowledge to help software engineers select the appropriate property templates. A

similar pattern-based approach to property specification was proposed by Dwyer, Avrunin, and Corbett in [20].

3.4.2 Property Translation

To support the integrated model/property translation, once the property specification language is defined, semantic entities for representing properties are introduced as extensions to the CAR profile for the source software language. A model and a property to be checked on the model are integrated in an instance of the CAR profile. In the xUML-to-S/R translation, properties are translated by a module of the translator. Since a property refers to semantic entities in the xUML model to be checked, this module conducts syntax and semantic checking on a property by referring to the abstract syntax tree constructed from the model. For each property template, a translation procedure is provided, which maps an instance of the template to the corresponding semantic entity in the CAR profile and ultimately to a property in S/R for use by COSPAN.

3.4.3 Automatic Generation of Properties

Certain types of properties, such as safety properties that check buffer overflows, can be automatically generated during translation. Translators can apply static analysis techniques that identify implicit buffers and generate properties for checking possible overflows of these buffers. For instance, in xUML, every class instance has an implicit message buffer, which has the risk of buffer overflow. The xUML-to-S/R translator automatically generates a safety property for each message buffer. When the resulting S/R model is model checked, the safety property will catch any buffer

overflow related to the message buffer being monitored. Automatically generated properties are integrated into translation in the same way as user-defined properties.

3.4.4 Translation Support for Compositional Reasoning

Another application of the software level property specification language is in constructing abstractions of components to be used in compositional reasoning [19] where model checking a property on a system is accomplished by decomposing the system into components, checking component properties locally on the components, and deriving the property of the system from the component properties. A property of a component is model-checked on the component by assuming that a set of properties hold on other components in the system. These assumed properties are abstractions of other components in the system and are used to create the closed system on which the property of the target component to be verified is model checked. These properties are formulated in the software level property specification language. The assumed properties on other components are called the *environment assumptions* of the target component. To support compositional reasoning, the translator is required to support translation of a closed system that consists of a component of a system and the environment assumptions of the component. This is in contrast to model checking without compositional reasoning where the translator is only required to translate a closed system that consists purely of entities specified in the software language, xUML in our case.

3.5 Transformations for State Space Reduction

The ultimate goal of integrated model/property translation is to generate a formal model which preserves only the behaviors of the source software model required for model checking a specific property and which has a minimal state space. Many state space reduction algorithms can be implemented as source-to-source transformations in the translation. This section describes model transformations implemented in the xUML-to-S/R translation and a model annotation language used to specify some types of transformations. Similar transformations will surely be applied in translation from most software specification languages to directly model-checkable formal languages.

3.5.1 Model Annotation Languages

There is often domain-specific information that is not available in a software model, but can facilitate transformations for state space reduction, for instance, bounds for variables in the software model. Software engineers can introduce such information by annotating the model with an annotation language before the model is translated. Such annotations are introduced in an xUML model as comments with special structures so that they will not affect other tools for xUML, for instance, xUML model execution simulators. The annotations must be updated accordingly as the model is updated.

Variable bounds are introduced in an xUML model as annotations associated with the variables or the data types of the variables. Annotation-based variable bounding indirectly enables symbolic model checking with COSPAN and also directly reduces state spaces. If tight bounds can be provided for variables in a soft-

ware model, it can often significantly reduce the state space of the resulting formal model that is to be explored by either an explicit state space enumeration algorithm or a symbolic search algorithm. Model checking guarantees the consistency among variable bounds by automatically detecting any possible out-of-bound variable assignments. The annotation language is also used to specify directives for guiding the loop abstraction [59].

Model annotations not only enable transformations, but also are indispensable to translation of continuous or infinite semantic entities in a software model. For instance, in the xUML-to-S/R translation, the information about how to discretize a float type and about the bounds for message buffers of class instances is also provided as annotations.

3.5.2 Transition Compression

A sequence of transitions in a software model can often be compressed and translated into a single transition in the formal model if verification of the property does not require intermediate states in the sequence. A transition compression algorithm can be generic, i.e., can be applied to many software languages, or language-specific, i.e, utilizes language-specific information to facilitate transition compression.

Generic transition compression

We use a simple example to illustrate generic transition compression. Suppose a simple program segment is of the form $x = 1;\ x = x + 1$. If a property to be checked is not relevant to the interleavings of the two statements with statements from other program segments, to the interim state between the two statements, or

to the variable, x, the program segment can be compressed into a single statement $x = 2$ without affecting the model checking result. Similar transition sequences appear in almost all programs in various software specification languages. Detailed discussions on generic transition compression can be found in [40].

Language-specific transition compression

There will be language-specific opportunities for transition compression in most software specification languages. An illustration of language-specific transition compression in the xUML-to-S/R translation is the identification and translation of self-messages. A self-message is a semantic feature specific to xUML and some other message-passing semantics: a class instance can send itself a message so that it can move from its current state to some next state according to a local decision. (It is assumed that self-messages have higher priority than other messages.) Sending and consuming of a self-message can be translated in a similar way as how sending and consuming of common messages among class instances are translated. This straightforward translation results in several S/R state transitions that simulate sending and consuming of a self-message. We developed a static analysis algorithm that identifies self-messages and translates sending and consuming of a self-message to a single S/R state transition.

3.5.3 Static Partial Order Reduction (SPOR)

Partial order reduction (POR) [25, 52, 62] is readily applicable to asynchronous interleaving semantics. POR takes advantages of the fact that in many cases, when components of a system are not tightly coupled, different execution orders of actions

or transitions of different components may result in the same global state. Then, under some conditions [25, 52, 62], in particular, when the interim global states are not relevant to the property being checked, model checkers need only to explore one of the possible execution orders. This may radically reduce model checking complexity.

The asynchronous interleaving semantics of xUML suggests application of POR. POR is applied to an xUML model through SPOR [39], a static analysis procedure that transforms the model prior to its translation into S/R by restricting its transition structure with respect to a property to be checked. For different properties, an xUML model may be translated to different S/R models if SPOR is applied in translation. Application of symbolic model checking to an S/R model translated from an xUML model transformed by SPOR enables integrated application of POR and symbolic model checking.

3.5.4 Predicate Abstraction

Predicate abstraction [26] maps the states of a concrete system to the states of an abstract system according to their evaluation under a finite set of predicates. Predicate abstraction is currently applied in model checking of software designs in xUML by application of the predicate abstraction algorithms proposed in [49] to the S/R models translated from these designs. It should be possible, however, to implement some forms of predicate abstraction as transformations in translation. Research on application of predicate abstraction to software system designs as they are translated is in progress.

3.6 Translator Validation and Evolution

Correctly model checking a software model through translation depends on correctness of (1) the conceptual semantics mapping from the source software language to the target formal language, (2) the translator that implements the semantics mapping, and (3) the underlying model checker that checks the resulting formal model. Correctness of a semantics mapping can sometimes be proved rigorously. A proof for the semantics mapping from xUML to S/R can be found in [67]. The translator must be validated to ensure that it correctly implements the translation from the source language to the target language and also the state space reduction algorithms incorporated. The correctness of the model checker is out of the scope of this chapter. As the source language and the target language evolve, the translator must also evolve to handle (or utilize, respectively) semantic entities that are newly introduced to the source (or target) language. The translator also must evolve to incorporate new state space reduction algorithms.

3.6.1 Translator Validation

Testing is the most commonly used method for validating a translator. Testing of a translator is analogous to, but significantly different from, testing of a conventional compiler. Testing of a conventional compiler is most often done by use of a suite of programs which are intended to cover a wide span of programs and paths through the compiler. Testing of a translator from a software specification language to a model-checkable formal language is a multi-dimensional problem. The test suite must be a cross-product of models, properties, and selections of state space reduction transformations. The correctness of a compilation can be validated by

running the compiled program for a spectrum of inputs and initial conditions and determining whether the outputs generated conform to known correct executions. While a translated model can be model checked, it is far more difficult to generate a suite of models and properties for which it is known whether or not a property holds on a model. We have a partial test suite for the xUML-to-S/R translation and development of a systematic test suite is in progress. Development of test suites is one of the most challenging problems faced by developers of translation-based model checking systems. We believe this is a problem which requires additional attention.

Recently, there has been progress on formal validation of the correctness of translators. The technique of *translation validation* is proposed in [54], whose goal is to check the result of each translation against the source program and thus to detect and pinpoint translation errors on-the-fly. This technique can improve, however cannot entirely replace the testing approach discussed above since the correctness of translation validation depends on the correctness of the underlying proof checker.

3.6.2 Translator Evolution

The key to the evolution of a translator is the evolution of the CAR of the translator since the CAR bridges the source software language to the target formal language and connects the translator frontend to the translator backend. Translation from the source language to the CAR is relatively straightforward since the CAR is quite simple. The complexity of the translation from the CAR to the target model-checkable language depends on the complexity of the target language, but the latter are also usually simple and well structured. The transformations conducted on the CAR are much more complex. The principle for the CAR evolution is that the

CAR should be kept stable as possible, and existing translation algorithms and state space reduction algorithms should be reused as much as possible. The CAR is extended (1) if there is no efficient way to translate some semantic entities of a new source language, (2) if some semantic entities of a new target language are hard to utilize, or (3) if implementation of new state space reduction algorithms requires introduction of new semantic entities in the CAR.

3.7 ObjectCheck Toolkit

To provide comprehensive automation support for model checking of xUML models, we have constructed the ObjectCheck toolkit that encapsulates the xUML-to-S/R translator. ObjectCheck supports xUML level property specification, xUML-to-S/R translation and optimization, error report generation, and error visualization. The architecture of ObjectCheck is shown Figure 3.2, under which we selected industrial tools such as Bridgepoint [56] or Objectbench [58], as the xUML Integrated Development Environment (IDE). Furthermore, we implemented the following components:

Property Specification Interface. The property specification interface enables formulation of properties to be checked on xUML models using the property specification language introduced in Section 3.4.1.

Error Report Generator. When an S/R query fails on an S/R model, COSPAN generates an error track specifying an execution trace that violates the query. The error report generator compiles an error report in xUML notations from the error track. The error report consists of an execution trace of the corresponding xUML model, which violates the corresponding xUML level property.

Error Visualizer. To facilitate debugging an error found by COSPAN in an xUML

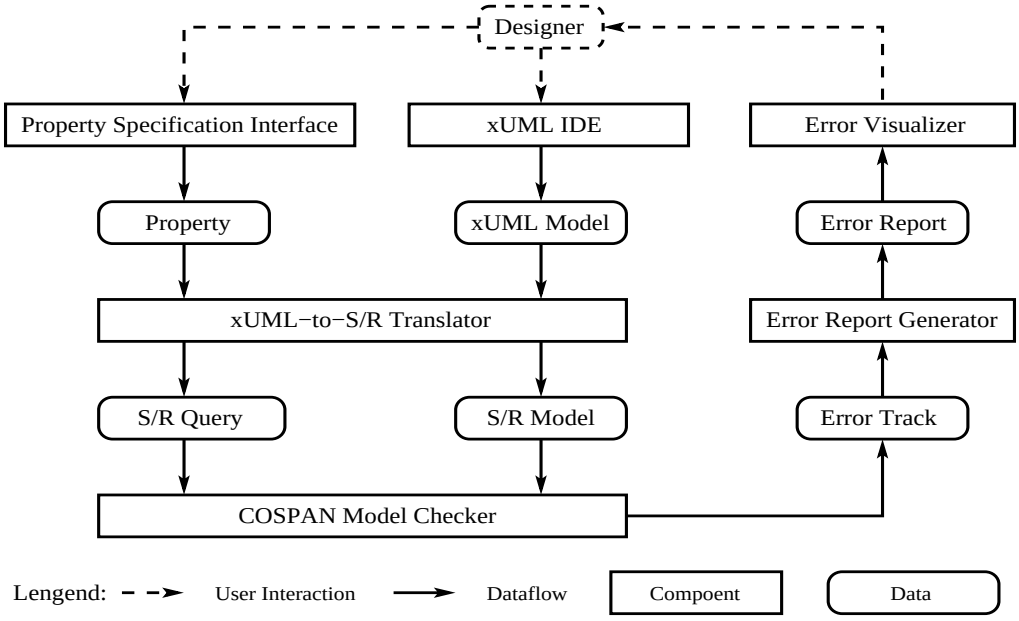

Figure 3.2: ObjectCheck architecture

model, an error visualizer is provided, which generates a test case from the error report and reproduces the error by executing the xUML model with the test case in a simulator included in the xUML IDE.

3.8 Case Studies Using ObjectCheck Toolkit

The ObjectCheck toolkit has been applied in model checking the design models of real-world software systems: a robot control system [36] from the robotics research group at the University of Texas at Austin, a prototype online ticket sale system [64], the TinyOS run-time system [30] for networked sensors from University of California, Berkeley. The robot control system case study, presented in [60], demonstrates model checking of non-trivial software design models with the xUML-to-S/R trans-

lator. In online ticket sale system case study, presented in Section 5.4.3, state space reduction capabilities of model transformations in the xUML-to-S/R translator and interactions of these transformations have been investigated. The TinyOS case study, presented in Section 6.4, demonstrates the translation support for compositional reasoning. Co-design and co-verification studies on TinyOS using the xUML-to-S/R translator are in progress.

3.9 Related Work

Most automatic approaches to model checking of design level software specifications are based on translation. Translators have been implemented for various design level specification languages such as dialects of UML, SDL, and LOTOS [6]. The vUML tool [43] translates a dialect of UML into Promela. The translation is based on ad-hoc execution semantics which did not include action semantics, and does not support specification of properties to be checked on the UML model level. There is also previous work [24, 48] on verification of UML Statecharts by translating Statecharts into directly model-checkable languages. The CAESAR system [22] compiles a subset of LOTOS into extended Petri nets, then into state graphs which are then model-checked by using either temporal logics or automata equivalences. The IF validation environment [8] proposes IF [7], an intermediate language, and presents tools for translating dialects of UML and SDL into IF and tools for validation and verification of IF specifications.

The translator architecture presented in this chapter extends the architecture for conventional compilers. Similar extensions have been proposed in [22, 8]. In these architectures, intermediate representations that have fixed and complete

semantics are adopted while in our approach, the CAR does not have fixed and complete semantics. It only specifies semantics of the generally shared semantic entities and for other semantic entities, their semantics are decided when a CAR profile is defined for a specific source language. This enables reuse of translator development efforts while allowing flexible translator development via a customizable intermediate representation.

A recent approach to model checking implementation level software representations is an integrated approach based on abstraction and translation. Given a program in C/C++ or Java, an abstraction of the program is created with respect to the property to be checked. This abstraction is constructed in a conservative way, i.e., if the property holds on the abstraction, the property also holds on the program. The abstraction is then translated into a model-checkable language and model checked. If the property does not hold on the abstraction, the error trace from model checking the abstraction is used to determine if the error is introduced by the abstraction process. If so, the abstraction is refined based on the error trace. The SLAM [5] tool from Microsoft, the FEAVER [33] tool from Bell Labs, and the Bandera [18] tool from Kansas State University are sample projects of this approach. SLAM abstracts a boolean program from a C program, then directly model-checks the boolean program or translates the boolean program into other model-checkable languages. FEAVER abstracts a state machine model from a C program with user help and translates the state machine model into Promela. Bandera abstracts a state machine model from a Java program and translates the state machine model into Promela, SMV, and other model-checkable languages. Many of translation issues identified in our project also appear in the translation phase of these three tools.

3.10 Summary

Translation plays an increasingly important role in software model checking and enables reuse of mature model checking techniques. This chapter identifies and formulates issues in translation of executable software designs for model checking. Solutions to these issues are presented in the context of the xUML-to-S/R translator. These solutions can be adapted to address similar issues in translation of other design level or implementation level software representations for model checking.

Chapter 4

Translation-Based Compositional Reasoning

4.1 Motivation and Overview

On account of the intrinsic computational complexity of model checking, we need to support compositional reasoning [53, 1, 3, 46, 4, 19] where model checking a property on a system is accomplished by decomposing the system into components, checking component properties locally on the components, and deriving the property of the system from the component properties. Application of compositional reasoning to software systems requires establishing a compositional reasoning rule in the semantics of these systems, proving the correctness of the rule, and implementing the rule. A rule is implemented when methods have been provided for discharging its premises which are usually verification of component properties, validity check of possible circular dependencies among component properties, and derivation of a

system property from component properties.

Directly proving the correctness of compositional reasoning rules for software systems is often difficult. Software systems are usually modeled in specification languages such as Executable UML [47] and SDL [35], or coded in programming languages such as Java and C/C++. These languages are sufficiently complicated in syntax and semantics so that it is very difficult (if not infeasible) to directly prove for these languages that a compositional reasoning rule is sound. Additionally, such a language often has varying operational semantics. A formal semantics is only formulated for software systems specified in this language when these systems are to be translated into a model-checkable formalism and verified. On the other hand, proof and implementation of compositional reasoning rules for directly model-checkable formal semantics such as the semantics of Promela [31], SMV [45], and S/R [28] is often easier due to the formality and simplicity of these semantics. It is often the case that a set of compositional reasoning rules have already been proven and implemented for these semantics.

This chapter defines, describes, and illustrates Translation-Based Compositional Reasoning (TBCR), an approach to application of compositional reasoning in the context of model checking software systems through model translation. This approach has two phases: (i) establishment of compositional reasoning rules in the semantics of software systems and correctness proof of the rules; (ii) application of the proven rules in model checking software systems. Given a translation from a software semantics to a directly model-checkable formal semantics, a compositional reasoning rule in the software semantics is established and proven for correctness as follows:

- The compositional reasoning rule is defined in the software semantics.
- The rule in the software semantics is mapped to an equivalent rule in the formal semantics based on the translation.
- The correctness proof of the rule is established based on the above mapping and on the correctness proof of the equivalent rule in the formal semantics.

Given a software system and a property to be checked on the system, the proven compositional reasoning rule in the software semantics is then applied as follows:

- The system is decomposed into components on the software semantics level.
- Premises of the rule are formulated in the software semantics. These premises are discharged by translating them to their counterparts in the formal semantics and discharging their counterparts in the formal semantics through reusing the implementation of the equivalent rule in the formal semantics.
- If these premises are successfully discharged, then it can be concluded on the software semantics level that the system has the property to be checked.

There has been a large body of research [53, 1, 3, 46, 4, 19] on compositional reasoning in the formal methods community, which mostly focuses on developing compositional reasoning rules and proving their correctness. Our research, instead, focuses on effective application of compositional reasoning to software systems in the context of model checking these systems via model translation. Rationales for our approach are:

- Software systems, to be model checked, usually have to be translated into a directly model-checkable formalism.

- Formulation of and reasoning about the properties of software systems and their components are more naturally accomplished in the software semantics.
- Compositional reasoning rules have already been established, proven, and implemented for several directly model-checkable formalisms.

We have realized TBCR for a commonly used semantics for software, the Asynchronous Interleaving Message-passing (AIM) semantics. In this realization, compositional reasoning rules in the AIM semantics are proven, implemented, and applied in the context of a translation from the AIM semantics to the ω-automaton semantics [37] using the I/O-automaton semantics [44] as an intermediate semantics. (We choose I/O-automata as the intermediate semantics to reuse a translation from the I/O-automaton semantics to the ω-automaton semantics, established by Kurshan, Merritt, Orda, and Sachs [41].) This realization has been applied in an integrated state space reduction framework [65] and in model checking of component-based software systems [66].

The balance of this chapter is organized as follows. In Section 4.2, we give the preliminaries of the I/O-automaton semantics and the ω-automaton semantics. A realization of TBCR for the AIM semantics is defined and described in detail in Section 4.3. We summarize in Section 4.4.

4.2 Preliminaries

4.2.1 I/O-automaton Semantics

The following definitions for I/O-automaton are from [41].

Definition 4.2.1 *An I/O automaton A is a quintuple $(\Sigma^A, S^A, I^A, \delta^A, R^A)$ where:*

- *the signature* Σ^A *is a triple* $\Sigma^A = (\Sigma^A_{IN}, \Sigma^A_{OUT}, \Sigma^A_{INT})$*, where* Σ^A_{IN}*,* Σ^A_{OUT}*,* Σ^A_{INT} *are pairwise disjoint finite sets of elements, called input, output, internal actions, respectively. We denote by* $\Sigma^A_{EXT} = \Sigma^A_{IN} \cup \Sigma^A_{OUT}$ *the set of external actions, by* $\Sigma^A_{LOC} = \Sigma^A_{OUT} \cup \Sigma^A_{INT}$ *the set of local actions, and we abuse notation, denoting by* Σ^A *also the set of all actions* $\Sigma^A_{LOC} \cup \Sigma^A_{IN}$*;*

- S^A *is a finite set of states;*

- $I^A \subset S^A$ *is a set of initial states;*

- $\delta^A \subset S^A \times \Sigma^A \times S^A$ *is a transition relation which is complete in the sense that for all* $a \in \Sigma^A_{IN}$*,* $s \in S^A$ *there exists* $s' \in S^A$ *with* $(s, a, s') \in \delta^A$*. For* $a \in \Sigma^A_{LOC}$ *and* $s, s' \in S^A$ *such that* $(s, a, s') \in \delta^A$*, we say that* a *is enabled at* s *and enables the transition* (s, s')*; Each element of* δ^A *is called a step of* A*;*

- R^A *is a partition of* Σ^A_{LOC}*, each element of which is termed a fairness constraint of* A*.*

Definition 4.2.2 *An execution of* A *is a finite string or infinite sequence of state-action pairs* $((s_1, a_1), (s_2, a_2), \ldots)$*, where* $s_1 \in I^A$ *and for all* i*,* $s_i \in S^A$*,* $a_i \in \Sigma^A$ *and* $(s_i, a_i, s_{i+1}) \in \delta^A$*.*

Definition 4.2.3 *An execution* $\mathbf{x}$ *of* A *is fair if, for all* $C \in R^A$*:*

- *if* $\mathbf{x}$ *is finite then no action in* C *is enabled in the final state in* $\mathbf{x}$*;*

- *if* $\mathbf{x}$ *is infinite then either some action in* C *occurs infinitely often in* x *or else infinitely many states in* $\mathbf{x}$ *have no enabled action which is in* C*.*

Definition 4.2.4 *Given a set $\triangle \subset \Sigma^A$, the projection of an execution $\mathbf{x}= ((s_i, a_i))$ of A onto $\triangle$, denoted by $\Pi_\triangle(\mathbf{x})$, is the subsequence of actions obtained by removing from the action sequence (a_i) all actions $a_i \notin \triangle$.*

Definition 4.2.5 *A behavior of A is the projection of a fair execution of A on the set Σ^A_{EXT} (i.e., the fair execution, with states and internal actions removed). The language $\mathcal{L}(A)$ of A is the set of all behaviors of A.*

Definition 4.2.6 *Of two I/O automata A and B, we say that A implements B (denoted by $A \leq B$) if, for $\triangle = \Sigma^A_{EXT} \cap \Sigma^B_{EXT}$, $\triangle \neq \emptyset$, $\Pi_\triangle(\mathcal{L}(A)) \subset \Pi_\triangle(\mathcal{L}(B))$.*

Definition 4.2.7 *For I/O automata $A_1, A_2, \ldots, A_k$, with respective pairwise disjoint sets of local actions, their parallel composition, denoted by $A_1||A_2||\ldots||A_k$, is an I/O automaton A defined as follows. The set of internal actions of A is the union of the respective sets of internal actions of the component automata, and likewise for the output actions; the input actions of A are the remaining actions of the components not thus accounted for. The set of states of A, S^A, is the Cartesian product of the component state sets, likewise for the initial states I^A. The transition relation δ^A is defined as follows: for $s = (s_1, \ldots, s_k)$, $s' = (s'_1, \ldots, s'_k)$ and $a \in \Sigma^A$, $(s, a, s') \in \delta^A$ if and only if for all $i = 1, \ldots, k$, $(s_i, a, s'_i) \in \delta^{A^i}$ or $a \notin \Sigma^{A^i}$ and $s'_i = s_i$. R_A is the union of the fairness partitions of the respective components.*

4.2.2 ω-automaton Semantics

We use the L-process model of ω-automaton semantics. Detailed specification of this model can be found in [37]. The concepts essential for understanding this chapter are given below for the convenience of the reader.

Definition 4.2.8 *For an L-process, ω, its language, $\mathcal{L}(\omega)$, is the set of all infinite sequences accepted by ω.*

Definition 4.2.9 *For L-processes, $\omega_1, \ldots, \omega_n$, their synchronous parallel composition $\omega = \omega_1 \otimes \ldots \otimes \omega_n$, is also an L-process and $\mathcal{L}(\omega) = \cap\mathcal{L}(\omega_i)$.*

Definition 4.2.10 *For L-processes, $\omega_1, \ldots, \omega_n$, their Cartesian sum, $\omega = \omega_1 \oplus \ldots \oplus \omega_n$, is also an L-process and $\mathcal{L}(\omega) = \cup\mathcal{L}(\omega_i)$.*

For a language, $\mathcal{L}$, let $\mathcal{CL}(\mathcal{L})$ denote the safety closure [2] of $\mathcal{L}$. [1]

Definition 4.2.11 *The safety closure $CL^{\omega}(\omega)$ of an L-process ω is an L-process whose language is the safety closure of the language of ω, $\mathcal{L}(CL^{\omega}(\omega)) = \mathcal{CL}(\mathcal{L}(\omega))$.*

Given an L-process ω, $CL^{\omega}(\omega)$ can be derived from ω by computing the Strong Connected Components (SCCs) of the state graph of ω and for each SCC with an accepting state, marking every state of that SCC as accepting.

Under the ω-automaton semantics model checking is reduced to checking L-process language containment. Suppose a system is modeled by the composition $\omega_1 \otimes \ldots \otimes \omega_n$ of L-processes, $\omega_1, \ldots, \omega_n$, and a property to be checked on the system is modeled by an L-processes, ω. The property holds on the system if and only if the language of $\omega_1 \otimes \ldots \otimes \omega_n$ is contained by the language of ω, $\mathcal{L}(\omega_1 \otimes \ldots \otimes \omega_n) \subset \mathcal{L}(\omega)$.

Definition 4.2.12 *Given two L-processes ω_1 and ω_2, ω_1 implements ω_2 (denoted by $\omega_1 \preceq \omega_2$) if $\mathcal{L}(\omega_1) \subset \mathcal{L}(\omega_2)$.*

[1] For a language $\mathcal{L}$ of sequences over a set of variables, V, the safety closure of $\mathcal{L}$, denoted by $\mathcal{CL}(\mathcal{L})$, is defined as the set of sequences over V where $x \in \mathcal{CL}(\mathcal{L})$ if and only if for all $j < |x|$ there exists y such that $x[0..j] : y$ belongs to $\mathcal{L}$ [4]. ($|x|$ denotes the length of x and $x : y$ denotes the concatenation of x and y where x and y are sequences over V.) In [37], $\mathcal{CL}(\mathcal{L})$ is termed as the smallest limit prefix-closed language that contains $\mathcal{L}$.

4.3 Realization of TBCR for AIM Semantics

This section presents how TBCR is realized for the AIM semantics. First, we informally describe the AIM semantics. Then, we formalize the AIM semantics, which enables the establishment, correctness proof, implementation, and application of compositional reasoning rules. After that, we describe how a compositional reasoning rule for the AIM semantics is established. Then, we prove this rule based on a translation from the AIM semantics to the ω-automaton semantics using the I/O-automaton semantics as an intermediate semantics. Finally, we present the implementation of this rule through the translation from the AIM semantics to the ω-automaton semantics.

4.3.1 Informal Description of AIM Semantics

Under the AIM semantics, a system is a composition of processes which interact asynchronously via message-passing. Every process has a private message queue and locally defined variables. Behaviors of a process are captured by an extended Moore state model and each state in the state model may have an associated state action that is composed from executable statements such as an assignment statement, a messaging statement, and an "if" statement. At any given moment of a system execution, there is exactly one process that is executing either a state action or a state transition in a run-to-completion fashion.

4.3.2 Formalization of AIM Semantics

A state in the extended Moore state model of an AIM process represents a set of states in the state space of the process. A state action in the extended Moore

state model represents multiple sequences of state transitions in the state transition structure of the process. To formally represent the extended Moore state model, we introduce a variable, *pc*, whose current value captures the current state in the Moore state model and the current position in the state action associated with the state. The message queue of the process is also formally represented by a variable, *queue*, whose domain includes all possible message permutations that may appear in the queue. Under this representation of message queues, the execution of a messaging statement in a process modifies the *queue* variable of the receiver process. With the above representations, we formally define an AIM process.

Definition 4.3.1 *An AIM process, P, is a six-tuple, (S, I, M, E, T, F), where:*

- *S, the state space of P, is the Cartesian product of the domains of the variables defined in the process and the two additional variables, pc and queue.*
- *I is a set of initial states.*
- *M is a messaging interface which is a pair, (M^i, M^o), where M^i is the set of messages that P inputs and M^o is the set of messages that P outputs.*
- *E is a set of events each of which is a state transition of the Moore state model, or an executable statement (such as an assignment statement, a messaging statement sending a message defined in M^o, or an "if" statement), or a reception of a message defined in M^i. E_{LOC} is a subset of E including all state transitions and executable statements in E. E_{EXT} is a subset of E including all messaging statements and message receptions in E.*
- *T is a set of state transitions defined on S and E, each of which is of the form, $(s,\ e,\ s')$, where $s,\ s' \in S$ and $e \in E$.*

- *F is a partition of E_{LOC}. Each element of F is termed a fairness constraint.*

Definition 4.3.2 *An execution of P is a finite string or an infinite sequence of state-event pairs ($(s_0,\ e_0)$, $(s_1,\ e_1)$, ...) which conforms to the run-to-completion requirement (i.e., the action statements from a state action appear adjacently in the execution), where $s_0 \in I$ and for all i, $s_i \in S$, $e_i \in E$ and $(s_i,\ e_i,\ s_{i+1}) \in T$. Fair executions of P are defined analogously to fair executions of an I/O-automaton.*

Definition 4.3.3 *A behavior of P is the projection of a fair execution of P on E_{EXT} of P. The language of S, $\mathcal{L}(S)$, is the set of all behaviors of S.*

Definition 4.3.4 *Given two AIM processes P and Q, P implements Q (denoted by $P \models Q$) if for $\triangle = E_{EXT}(P) \cap E_{EXT}(Q)$ and $\triangle \neq \emptyset$, $\Pi_{\triangle}(\mathcal{L}(P)) \subset \Pi_{\triangle}(\mathcal{L}(Q))$.*

Definition 4.3.5 *The interleaving composition of a finite set of interacting AIM processes, P_0, P_1, ..., and P_n, denoted by $P_0[]P_1[]\ldots[]P_n$, is an AIM process, P, derived as follows. S is the Cartesian product of S_0, S_1, ..., and S_n. I is the Cartesian product of I_0, I_1, ..., and I_n. M^i includes the remaining messages in M_0^i, M_1^i, ..., and M_n^i that are not accounted for in the composition, and M^o is the union of M_0^o, M_1^o, ..., and M_n^o. E is the union of E_0, E_1, ..., and E_n. T is defined as follows: for $s = (s_0,\ s_1, \ldots,\ s_n)$, $s' = (s'_0,\ s'_1, \ldots,\ s'_n)$, and $e \in E$, $(s,\ e,\ s') \in T$ if and only if for all $i \in [0,\ n]$, $e \in E_i$ and $(s_i,\ e,\ s'_i)$ or $e \notin E_i$ and $s'_i = s_i$. F is the union of the fairness partitions of the respective components.*

In this formalized AIM semantics, a system, components of the system, and properties of the system and the components are all represented by processes.

4.3.3 Establishment of Compositional Reasoning Rules

We establish compositional reasoning rules for the AIM semantics by porting existing rules in directly model-checkable formal semantics to the AIM semantics. We have ported to the AIM semantics two rules that have already been established, proven, and implemented in the ω-automaton semantics, the rule proposed by Amla, Emerson, Namjoshi, and Trefler in [4], *Rule 1*, and the rule proposed by McMillan in [46]. Below we show how *Rule 1* is ported to the AIM semantics.

Rule 1 *For AIM processes P_1, P_2, and Q, to show that $P_1[]P_2 \models Q$, find AIM processes Q_1 and Q_2 such that the following conditions are satisfied.*

C1: $P_1[]Q_2 \models Q_1$ *and* $P_2[]Q_1 \models Q_2$

C2: $Q_1[]Q_2 \models Q$

C3: *Either* $P_1[]CL^P(Q) \models (Q + Q_1 + Q_2)$ *or* $P_2[]CL^P(Q) \models (Q + Q_1 + Q_2)$

Let $P_1[]P_2$ denote a system composed from two components, P_1 and P_2. Q is a property to be checked on the system. Q_1 and Q_2 are properties of P_1 and P_2, respectively. Condition C1 checks if P_1 has the property, Q_1, assuming Q_2 holds on P_2, and if P_2 has the property, Q_2, assuming Q_1 holds on P_1. Condition C2 checks if Q can be derived from Q_1 and Q_2. Condition C3 conducts the validity check of circular dependencies between Q_1 and Q_2. (The counterpart of Rule 1 in the ω-automaton semantics, denoted by *Rule* 1^ω, is of the same form but with processes, $\models$, $[]$, CL^P, and $+$ replaced by their ω-automaton counterparts.)

To port compositional reasoning rules to the AIM semantics, additional semantics concepts may need to be introduced for the AIM semantics. In the case of

Rule 1, the concepts of safety closure of an AIM process and sum of AIM processes were defined:

Definition 4.3.6 *For an AIM process, Q, the safety closure of Q, $CL^P(Q)$, is an AIM process whose language is the safety closure [2] of the language of Q, $\mathcal{L}(CL^P(Q)) = \mathcal{CL}(\mathcal{L}(Q))$. ($CL^P(Q)$ can be derived from Q by removing the fairness constraints of Q.)*

Definition 4.3.7 *The Cartesian sum of AIM processes P and Q, denoted by $P + Q$, is the AIM process that behaves either as P or as Q and with the property of $\mathcal{L}(P + Q) = \mathcal{L}(P) \cup \mathcal{L}(Q)$.*

4.3.4 Proof via Semantics Translation

We first establish a translation from the AIM semantics to the ω-automaton semantics and then prove the soundness of Rule 1 based on the translation and the soundness proof of Rule 1^ω. To establish the translation from the AIM semantics to the ω-automaton semantics, we use the I/O-automaton semantics as an intermediate semantics.

Translation of AIM Processes to I/O-automata

An AIM process, P, is translated to an I/O-automaton, A, through a two-step procedure. The first step maps semantic constructs of P to semantic constructs of A and the second step implements the run-to-completion requirement in A.

Step 1: Mapping semantic constructs

- The state space and the initial state set of P are mapped to the state space and the initial state set of A correspondingly, which is achieved by mapping

the variables of P to the corresponding variables of A. (Note that the state space of an I/O automaton is also encoded by the domains of its variables.)

- Events of P are translated to actions of A as follows:
  - A state transition in the extended Moore state model of P is mapped to an internal action of A that simulates the state transition by modifying the variables, *pc* and *queue*, accordingly.
  - An assignment statement is mapped to an internal action that modifies the variable to be assigned by the assignment and the variable, *pc*.
  - An "if" statement is mapped to an internal action that modifies the variable, *pc*, to reflect the decision made in the "if" statement.
  - A messaging statement is mapped to an output action that is also an input action of the I/O-automaton corresponding to the receiver.
  - A message reception is mapped to an input action that modifies the variable, *queue*, and is also an output action of the sender I/O-automaton.
- Messages in the input (or output, respectively) interface of P are mapped to input (or output) actions of A.
- A state transition of P, $(s_P,\ e_P,\ s'_P)$, is mapped to a state transition of A, $(s_A,\ a_A,\ s'_A)$, where s_A, a_A, and s'_A are the corresponding translations of s_P, e_P, and s'_P as described above.

Step 2: Implementing run-to-completion requirement

- The I/O-automaton, A, resulting from Step 1 is extended with an additional boolean variable, *RtC*, and two output actions, *Enter* and *Leave*. The *Enter*

action cannot be enabled unless the value of *RtC* is false.

- When A is composed with A', the I/O-automaton translation of another AIM process, P', the *Enter* and *Leave* actions of A are included by A' as input actions and vice versa.

- The transition relation of A is extended so that before A executes the first I/O-automaton action in the sequence of I/O-automaton actions corresponding to a state action of P, A executes the *Enter* action and after A executes the last I/O-automaton action in the sequence of I/O-automaton actions corresponding to a state action of P, A executes the *Leave* action. (A' is extended in the same way.)

- The transition relation of A' is extended so that as A executes the *Enter* action, A' sets its *RtC* to true and as A executes the *Leave* action, A' sets its *RtC* to false and vice versa.

Therefore, when a set of I/O-automata translated from AIM processes are ready to execute their *Enter* actions, only one of them can proceed, execute its *Enter* action, and get into the run-to-completion section. The automaton signals its leaving the run-to-completion section by executing its *Leave* action. We refer to the translation from an AIM process to its corresponding I/O-automaton as T_A^P.

Theorem 4.3.1 *Given an AIM process,* $P = P_1[] \ldots []P_n$*, and its I/O automaton translation,* $A = T_A^P(P_1)|| \ldots ||T_A^P(P_n)$*, for* $\triangle = \Sigma_P^A$ *where* Σ_P^A *is the set of external actions of A excluding all Enter and Leave actions and* $\triangle \neq \emptyset$*,* $\mathcal{L}(P)$=$\Pi_\triangle \mathcal{L}(A)$.

Proof of Theorem 4.3.1: By the construction of A from P, $\mathcal{L}(P) = \Pi_\triangle \mathcal{L}(A)$. ■

Translation of AIM Processes to ω-automata

Kurshan, Merritt, Orda, and Sachs [41] have established a translation from I/O-automata to ω-automata, T^A_ω, and proved that the translation is linear-monotone with respect to language containment (shown in Theorem 4.3.2).

Theorem 4.3.2 *For two I/O-automata,* $A = A_1||\ldots||A_m$ *and* $B = B_1||\ldots||B_n$, $A \leq B \iff \mathcal{L}(T^A_\omega(A_1) \otimes \ldots \otimes \mathcal{L}(T^A_\omega(A_m)) \subset \mathcal{L}(T^A_\omega(B_1) \otimes \ldots \otimes \mathcal{L}(T^A_\omega(B_n))$.

Based on the translation from AIM processes to I/O-automata, T^P_A, and the translation from I/O-automata to ω-automata, T^A_ω, we constructed a translation from AIM processes to ω-automata, T^P_ω. For a given AIM process, P,

- P is first translated to an I/O-automaton $T^P_A(P)$;
- $T^P_A(P)$ is then translated to an ω-automaton $T^A_\omega(T^P_A(P))$.

We demonstrate with Theorem 4.3.3 that T^P_ω is also linear-monotone with respect to language containment.

Theorem 4.3.3 *For two AIM processes,* $P = P_1[]\ldots[]P_m$ *and* $Q = Q_1[]\ldots[]Q_n$, $P \models Q \iff \mathcal{L}(T^P_\omega(P_1) \otimes \ldots T^P_\omega(P_m)) \subset \mathcal{L}(T^P_\omega(Q_1) \otimes \ldots T^P_\omega(Q_n))$.

Proof of Theorem 4.3.3: Follows directly from Theorem 4.3.1 and Theorem 4.3.2. ■

Lemma 4.3.1 *For an AIM process P,* $CL^\omega(T^P_\omega(P)) \preceq T^P_\omega(CL^P(P))$.

Proof of Lemma 4.3.1:

$\Rightarrow$ {Definition 4.3.6, Definition 4.3.4}

$$
\begin{aligned}
& P \models CL^P(P) \\
\Rightarrow\ & \{\text{Theorem 4.3.3}\} \\
& \mathcal{L}(T^P_\omega(P)) \subset \mathcal{L}(T^P_\omega(CL^P(P))) \\
\Rightarrow\ & \{\text{Monotonicity of language closure}\} \\
& \mathcal{CL}(\mathcal{L}(T^P_\omega(P))) \subset \mathcal{CL}(\mathcal{L}(T^P_\omega(CL^P(P)))) \\
\Rightarrow\ & \{\text{Definition 4.2.11}\} \\
& \mathcal{L}(CL^\omega(T^P_\omega(P))) \subset \mathcal{CL}(\mathcal{L}(T^P_\omega(CL^P(P)))) \\
\Rightarrow\ & \{\text{A safety property is the safety closure of itself.}\} \\
& \mathcal{L}(CL^\omega(T^P_\omega(P))) \subset \mathcal{L}(T^P_\omega(CL^P(P))) \\
\Rightarrow\ & \{\text{Definition 4.2.12}\} \\
& CL^\omega(T^P_\omega(P)) \preceq T^P_\omega(CL^P(P))
\end{aligned}
$$

■

Lemma 4.3.2 *For AIM processes* $P_1, \ldots, P_n$, $T^P_\omega(P_1 + \ldots + P_n) \preceq T^P_\omega(P_1) \oplus \ldots \oplus T^P_\omega(P_n)$.

Proof of Lemma 4.3.2: Follows directly from Definition 4.3.7, Theorem 4.3.3, Definition 4.2.10, and Definition 4.2.12. ■

Theorem 4.3.4 *Rule 1 is sound for arbitrary AIM processes,* P_1, P_2, *and* Q.

Proof Sketch of Theorem 4.3.4: Suppose Conditions C1, C2, and C3 hold on P_1, P_2, and Q. Due to Theorem 4.3.3, Lemma 4.3.1, and Lemma 4.3.2, the counterparts of Conditions C1, C2, and C3 in the ω-automaton semantics hold on $T^P_\omega(P_1)$, $T^P_\omega(P_2)$, and $T^P_\omega(Q)$. Therefore, by Rule 1^ω (the counterpart of Rule 1 in

the ω-automaton semantics), $T^P_\omega(P_1) \otimes T^P_\omega(P_2) \preceq T^P_\omega(Q)$. By Theorem 4.3.3, we conclude that $P_1[]P_2 \models Q$. (Detailed proof of this theorem can be found in the appendix.) ■

4.3.5 Implementation and Application via Model Translation

TBCR suggests that a compositional reasoning rule in the AIM semantics be implemented based on the translation from the AIM semantics to the ω-automaton semantics and by reusing the implementation of its equivalent rule in the ω-automaton semantics. We first introduce an implementation of the AIM-to-ω-automaton translation and an implementation of Rule 1^ω (the ω-automaton semantics counterpart of Rule 1) in the ω-automaton semantics. We then discuss how Rule 1 is implemented and applied.

Translation from xUML to S/R

xUML [47] is an executable dialect of UML whose semantics conforms to the AIM semantics given in this chapter. S/R [28] is an automaton language whose semantics conforms to the ω-automaton semantics. In Chapter 3, we have presented a translator from xUML to S/R. Given a system modeled in xUML and a property specified in an xUML level property specification language, the translator automatically translates the design and the property to an S/R model and an S/R property. The S/R property is then checked on the S/R model by the COSPAN [28] model checker. The property holds on the system if and only if the S/R property is successfully verified on the S/R model. As shown in Figure 4.1, the xUML-to-S/R translation syntactically translates an xUML model into S/R, which also implements the semantics

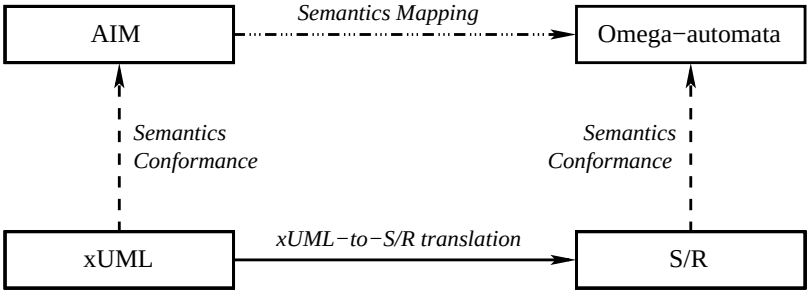

Figure 4.1: xUML-to-S/R translation implements the translation from AIM to ω-automata

mapping from the AIM semantics to the ω-automaton semantics.

Existing Implementation of Rule 1^{ω} in S/R

Rule 1^{ω} has been implemented in S/R [4]. Since in S/R, systems, components, assumptions, and properties are all modeled as ω-automata which can be trivially composed, verification of component properties (Condition C1) and derivation of a system property from component properties (Condition C2) are discharged in the same way as a property is checked on a system. Validation of circular dependencies (Condition C3) additionally requires construction of the safety closure of an ω-automaton (which has been discussed in Section 4.2.2).

Implementation and Application of Rule 1 in xUML

The xUML-to-S/R translator requires that an xUML model to be translated specify a closed system. To support Rule 1, the translator is extended to allow a closed system formed by a component of a system and its assumptions on the rest of the system (i.e. properties that the component assumes the rest of the system to

have). The extension is simplified by the fact that in S/R, systems, components, assumptions, and properties to be checked are all modeled as ω-automata which can be trivially composed. Based on the implementation of Rule 1^ω in S/R and the extended xUML-to-S/R translator, compositional reasoning using Rule 1 is applied in model checking software systems modeled in xUML as follows:

- Given a system in xUML and a property to be checked, the system is decomposed on the xUML level and premises of Rule 1 are formulated in xUML.
- These premises are discharged by translating them to their counterparts in S/R using the extended xUML-to-S/R translator and discharging their counterparts using the implementation of Rule 1^ω in S/R.

Correct application of Rule 1 then depends on the correctness of the translation from xUML to S/R and the correctness of the implementation of Rule 1^ω in S/R.

4.4 Summary

TBCR is a simple and effective approach to application of compositional reasoning in the context of model checking software systems through translation. It simplifies the correctness proof of compositional reasoning rules in software semantics and reuses existing proofs and implementations of compositional reasoning rules in directly model-checkable semantics. Its feasibility and effectiveness has been demonstrated by its realization for the AIM semantics and two applications of this realization: (1) in the integrated state space reduction framework, presented in Chapter 5, and (2) in the approach to component verification, presented in Chapter 6.

Appendix: Detailed Proof of Theorem 4.3.4

Proof of Theorem 4.3.4:

$$P_1[]Q_2 \models Q_1$$

$\Rightarrow$ {Theorem 4.3.3}

$$T_\omega^P(P_1) \otimes T_\omega^P(Q_2) \preceq T_\omega^P(Q_1) \tag{4.1}$$

$$P_2[]Q_1 \models Q_2$$

$\Rightarrow$ {Theorem 4.3.3}

$$T_\omega^P(P_2) \otimes T_\omega^P(Q_1) \preceq T_\omega^P(Q_2) \tag{4.2}$$

$$Q_1[]Q_2 \models Q$$

$\Rightarrow$ {Theorem 4.3.3}

$$T_\omega^P(Q_1) \otimes T_\omega^P(Q_2) \preceq T_\omega^P(Q) \tag{4.3}$$

$$P_1[]CL^P(Q) \models (Q + Q_1 + Q_2)$$

$\Rightarrow$ {Theorem 4.3.3}

$$T_\omega^P(P_1) \otimes T_\omega^P(CL^P(T)) \preceq (T_\omega^P(Q + Q_1 + Q_2))$$

$\Rightarrow$ {Lemma 4.3.1, Lemma 4.3.2}

$$T_\omega^P(P_1) \otimes CL^\omega(T_\omega^P(T)) \preceq (T_\omega^P(Q) \oplus T_\omega^P(Q_1) \oplus T_\omega^P(Q_2)) \tag{4.4}$$

$$P_2[]CL^P(Q) \models (Q + Q_1 + Q_2)$$

$\Rightarrow$ {Theorem 4.3.3}

$T^P_\omega(P_2) \otimes T^P_\omega(CL^P(Q)) \preceq (T^P_\omega(Q + Q_1 + Q_2))$

$\Rightarrow$ {Lemma 4.3.1, Lemma 4.3.2}

$$T^P_\omega(P_2) \otimes CL^\omega(T^P_\omega(Q)) \preceq (T^P_\omega(Q) \oplus T^P_\omega(Q_1) \oplus T^P_\omega(Q_2)) \tag{4.5}$$

$\{(1), (2), (3), (4), (5)\}$

$\Rightarrow$ {Rule 1^ω}

$T^P_\omega(P_1) \otimes T^P_\omega(P_2) \preceq T^P_\omega(Q)$

$\Rightarrow$ {Theorem 4.3.3}

$$P_1 [] P_2 \models Q \tag{4.6}$$

■

Chapter 5

Integrated State Space Reduction Framework

5.1 Motivation and Overview

Executable software system designs are ideal candidates for model checking due to their complete execution semantics and natural incorporation of state models. Their major features potentially enable effective state space reductions, for instance, compositional structures may lead to effective decompositions, inheritance relationships may facilitate abstractions, and multiple instances of a class may simplify the identification of symmetries.

This chapter defines and describes a general framework for integrated state space reduction in model checking executable software system designs. The framework assumes that the executable system designs can be translated into model-checkable languages and is discussed using system designs modeled in xUML. Under

the framework, state space reduction algorithms are applied in an integrated way to xUML models before and during the translation and to the resulting S/R models. Interactions among these algorithms are explored to maximize the aggregate effect of state space reduction.

Many software system designs are constructed following domain-specific design patterns that provide information about structures and behaviors of these systems. Reduction algorithms such as compositional reasoning [53, 1, 3, 46, 4, 19] abstraction [17], and symmetry reduction [16], whose effectiveness depends on structures and behaviors of software systems, can be readily formulated on design models due to the fact that execution behaviors of different components are more observable at the design level and due to the existence of domain-specific design patterns. State space reduction algorithms are often applied in combinations. These facts taken together suggest instantiating the general state space reduction framework for different application domains based on domain-specific design patterns.

Distributed transaction systems, which are commonly constructed in a design pattern of dispatchers, agents, and servers with customer initiated transactions as observable units of work, are examples of a family of systems for which a structured process for applying state space reduction algorithms at the design model level can be formulated. We illustrate the general framework with its instantiation for distributed transaction systems, a systematic process for reducing model checking a property on the design model of a transaction system to discharging a well-defined set of less complex model checking problems. The process represents a transaction as message sequences, associates the property to be checked with a transaction, partitions the model into sub-models, and decomposes the property

into sub-properties and assumptions defined over these sub-models. The process is evaluated by its application in model checking an online ticket sale system [64]. The dimension of transaction systems that can be model checked is materially extended by this process.

There has been extensive research on state space reduction algorithms for either hardware systems or software systems, which is surveyed in [17]. Our work, instead of focusing on particular state space reduction algorithms, explores the integrated application of reduction algorithms in the context of the general framework and investigates how domain-specific design patterns can help adapt the general framework to different application domains to achieve more automatic and effective state space reduction. Our work is distinguished from the integrated state space reduction for hardware systems [46] by focusing on software systems and incorporating both reduction algorithms effective for asynchronous semantics and those effective for synchronous semantics.

Section 5.2 defines the general framework, briefly describes the state space reduction algorithms applied in the context of the framework, and gives some guidelines for when to apply each state space reduction algorithm and for the application order of these algorithms. Section 5.3 sketches the partially implemented automation support for the general framework. Section 5.4 defines, describes, and illustrates the instantiation of the general framework on distributed transaction systems. Section 5.5 evaluates the instantiation with results from model checking an online ticket sale system. Section 5.6 summarizes.

5.2 Integrated State Space Reduction

In this section, a structured framework for integrated application of state space reduction algorithms to executable object-oriented software system designs is defined. The framework is presented for system designs modeled in xUML, but can be used to structure integrated state space reduction for other representations. The state space reduction algorithms being applied in the context of this framework are described and interactions among these algorithms are discussed.

5.2.1 General Framework

The model checking process for an xUML model, presented in Chapter 3, is a sequential application of the following two procedures:

- xUML-to-S/R translation that translates the xUML model and an xUML level property to be checked on the model to an S/R model and an S/R property;
- S/R level model checking that checks the S/R property on the S/R model by invoking the COSPAN model checker.

The process, referred to as the basic model checking process in Figure 5.1, works effectively on xUML models with small numbers of class instances, but cannot scale due to the state space explosion problem. On the other hand, for well-structured xUML models, there are system structure and property specific reduction algorithms at the xUML model level, which cannot be recognized by the xUML-to-S/R translator and the COSPAN model checker, but which can effect major state space reduction on the resulting S/R model that is to be model checked. Therefore, the general framework prefaces the basic model checking process with a user-driven state

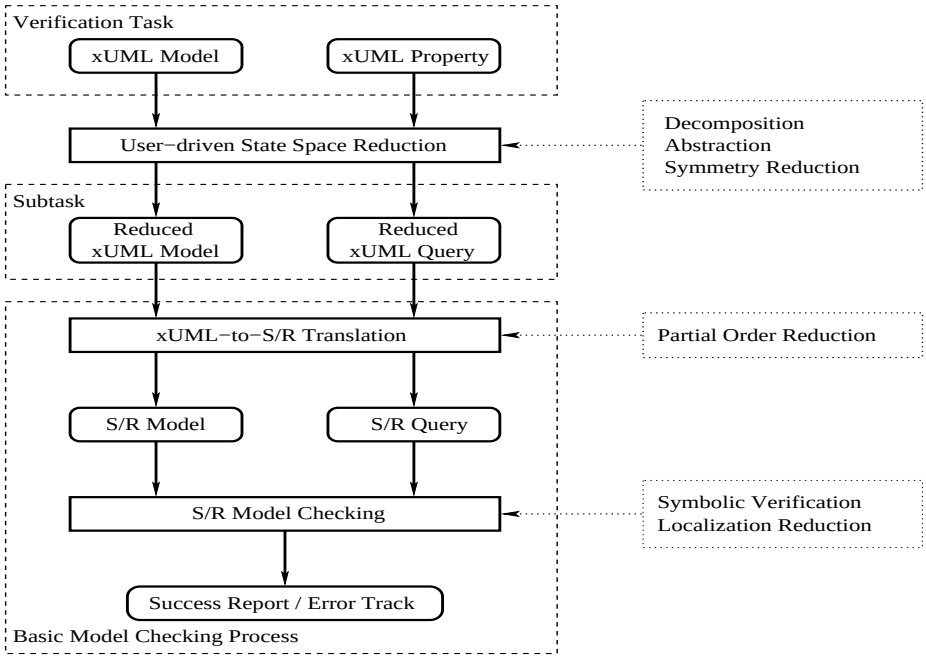

Figure 5.1: Reduction hierarchy of general framework

reduction procedure.

The general framework establishes a three-level hierarchy for integrated state space reduction, as shown in Figure 5.1. Different reduction algorithms are invoked on different levels of the hierarchy and applied to models of different forms:

- In the user-driven state space reduction procedure, user-driven reduction algorithms such as decomposition, abstraction, and symmetry reduction are applied to reduce a complex model checking task T, a complex property on a complex xUML model, into a set of subtasks. Each subtask checks a sub-property of the original property on a sub-model of the original model. A sub-model is either a component or an abstraction of the original model. Each

subtask is either discharged by invoking the basic model checking process or further reduced. The reductions applied are validated by invoking the basic model checking process or conducting a simple theorem proving.

- In the xUML-to-S/R translation procedure, automatic reduction algorithms, such as static partial order reduction, are applied, which transform an xUML model prior to its translation into S/R with respect to a given xUML property and construct an equivalent model that has a smaller state space.

- In the S/R level model checking procedure, automatic reduction algorithms implemented by COSPAN, such as symbolic model checking and localization reduction, are applied. These algorithms make use of the semantic information of an S/R model to reduce the state space to be explored by COSPAN.

Under the general framework, the extended model checking process for xUML models operates recursively and interactively as shown in Figure 5.2. A model checking task, T_0, is recursively reduced into subtasks. A reduction conjecture from users is always validated before its resulting model checking subtasks are discharged. The basic model checking process becomes a model checking engine for discharging subtasks. When a reduction conjecture or a subtask is verified to be false, user interaction is requested. Upon user inputs, either a new reduction conjecture is introduced, or the process is aborted.

5.2.2 Major State Space Reduction Algorithms

There are many possible state space reduction algorithms that can be applied to xUML models under the general framework. Some of them are summarized below.

```
Enqueue(ToDo, T_0); Done ={ }; /* ToDo is a queue and Done is a set. */
Do
  T = Dequeue(ToDo);
  If (T is Directly Model-Checkable) Then
    If (Basic-model-checking-process(T)) Then
      Done = Done + {T}; Continue;
    Else
      Error-report-generation(T); Invoke-user-interface ( );
  End;
  < T_1, ..., T_n > = User-driven-state-space-reduction(T);
  If (Valid(T, < T_1, ..., T_n >) Then Enqueue(ToDo, T_1, ..., T_n);
  Else
    Error-report-generation(T, < T_1, ..., T_n >); Invoke-user-interface( );
  End;
Until (Empty(ToDo));
```

Figure 5.2: Recursive and interative model checking process of general framework

Decomposition

The compositional hierarchy, the asynchronous message communication semantics, and the interleaving execution semantics of xUML make decomposition a natural state space reduction algorithm for xUML models.

- **Compositional Reasoning** [53, 1, 3, 46, 4, 19] The hierarchical structure of an xUML model may be explored to decompose the model into components that have simple and clear interfaces. A property on the xUML model can often be broken into a set of sub-properties on the model, its components, or its abstractions. Checking the sub-properties is simpler than checking the original property and verification of the sub-properties guarantees verification of the original property. Dependencies among components are formulated as assumptions of each component on other components. Therefore, a sub-property can be checked on a component under its assumptions, which

consumes less memory and time than checking the sub-property on the original model. Compositional reasoning is applied following the translation-based approach presented in Chapter 4.

- **Case Splitting** [46] In many xUML models, concurrent operations may often be grouped into units of work, for example, transactions in an e-business system. There may be little or no interaction among these units of work. If a property on the whole system can be decomposed into sub-properties on units of work and the units of work can be decoupled when these sub-properties are checked, significant state space reduction can often be achieved.

Abstraction

Three abstraction algorithms can be applied:

- **State Model Abstraction** [37] If a property is over one or several components of a system, state models in the components not directly involved in the property may be abstracted to reduce the state space to be explored for checking the property. If the abstraction is sound (Executions of the abstract system contain all behaviors of the original system.), then if the property is verified to be true on the abstract system, it will also be true on the original system. The most common form of state model abstraction is the non-deterministic abstraction. For instance, a decision point in a state model may be made non-deterministic and a set of state models that are only differentiated by their unique identifiers may be simulated by a state model with a non-deterministic identity. A major advantage of non-deterministic abstraction over other kinds of state model abstraction is that its correctness is automatically guaranteed.

- **Data Abstraction** [17] If a mapping can be found between data values of an xUML model and a small set of abstract data values, then an abstract xUML model that simulates the original model can be constructed by extending the mapping to states and transitions. Since the state space of the abstract model is usually smaller, it is often easier to check properties on the abstract model.

- **Localization Reduction** [37] Given a model and a property, localization reduction, also known as cone of influence reduction [17], eliminates variables in the model that do not influence the variables in the property. The checked property is preserved, but the size of the model to be checked is smaller.

Symmetry Reduction

Symmetry reduction can often reduce the number of properties to be checked on an xUML model or the state space size of the model.

- **Symmetric Property Reduction** [37] Given two properties on an xUML model, if a nontrivial mapping can be defined among variables in the model or among values of variables, which maps the model to itself and the two properties to each other, then only one of the two properties need to be checked on the model.

- **Quotient Model Reduction** [16] Having symmetry in a model implies the existence of nontrivial permutation groups that preserve both the state labeling and the transition relation. The quotient model induced by this relation is often smaller than the original model. Moreover it is bisimulation equivalent to the original model. Therefore, all properties on the original model can be instead checked on the quotient model.

Partial Order Reduction

Partial order reduction [25] [52] [62] takes advantages of the fact that, in many cases, when components of a system are not tightly coupled, different execution orders of actions or transitions of different components may result in the same global state. Then, under some conditions, in particular, when the interim global states are not relevant to the property being checked, model checkers only need to explore one of the possible execution orders. This may radically reduce model checking complexity.

Asynchronous interleaving semantics of xUML suggest application of static partial order reduction [39] to an xUML model prior to its translation into S/R, which transforms the xUML model by restricting its transition structure with respect to a property to be checked. This enables integrated application of partial order reduction while applying symbolic model checking to the S/R model.

Symbolic Model Checking

Symbolic model checking [45] represents the state transition structure of an xUML model with binary decision diagrams, which enables manipulation of entire sets of states and transitions instead of individual states and transitions. This heuristic is fully automatic and has shown encouraging reduction promise on some xUML models. (To be elaborated in Section 5.5).

5.2.3 Interactions among Reduction Algorithms

Under the general framework, state space reduction algorithms are applied to xUML models in an integrated way. To maximize the aggregate effect of state space reduction, the selection of reduction algorithms and the application order of the selected

reduction algorithms need to be carefully considered.

Selection of Reduction Algorithms

The structure of an xUML model and the knowledge of its execution behavior can help select the reduction algorithms to be applied to the model:

a. Symmetry reduction is often selected if there exist many instances of the same class;

b. Partial order reduction is often selected if there is intensive execution interleaving;

c. Symbolic model checking is often selected if there is much randomness.

d. Localization reduction is always applied to S/R models.

xUML models from different application domains, different xUML models from the same application domain, or different properties on the same xUML model may lead to different selections of reduction algorithms. Therefore, domain, model, and property specific knowledge need to be involved in the algorithm selection besides the selection guidelines provided.

Application Order of Reduction Algorithms

To maximize the state space reduction effect, it is always attempted to apply each reduction algorithm to the minimum models with which the algorithm has to deal. Therefore, the framework hard-codes some application ordering relations among reduction algorithms:

- Algorithms in the user-driven reduction procedure are always applied prior to algorithms in the xUML-to-S/R translation procedure.
- Algorithms in the xUML-to-S/R translation procedure are always applied prior to algorithms in the S/R level model checking procedure.
- In the S/R level model checking procedure, localization reduction is always applied prior to symbolic model checking.

There is no ordering relation defined among reduction algorithms applied in the user-driven reduction procedure because the ordering relations among these algorithms are also domain, model, and property specific.

5.2.4 Instantiation of General Framework for Application Domains

The framework defines a general process for structuring integrated state space reductions, but requires certain amount of user interaction. System designs from the same application domains commonly follow a set of domain-specific design patterns and require satisfaction of properties in similar formats. Therefore, domain-specific design patterns and property patterns can often be explored to establish an instantiation of the general framework for a given domain. The instantiation should provide additional guidelines for selecting reduction algorithms and additional relations for ordering these reduction algorithms. With these extra efforts, the instantiation may significantly reduce the user interaction required and make the integrated state space reduction for the given domain more automatic and effective. In Section 5.4, we demonstrate how the general framework is instantiated by instantiating it for distributed transaction systems.

5.3 Automation of Integrated State Space Reduction

Automation support, which is crucial to the wide application of the general framework, is provided by extending the xUML-to-S/R translator and introducing a reduction manager.

5.3.1 Extension to xUML-to-S/R Translator

The xUML-to-S/R translator was extended by incorporating the optimization module of SDLCheck [42] that implements static partial order reduction and other software-specific model checking optimizations. These optimizations transforms the xUML model with respect to the xUML property before the translation into S/R and can be switched on or off without affecting the translation.

5.3.2 Reduction Manager

A reduction manager has been designed and partially developed, which coordinates the recursive model checking process in Figure 5.2. If the current subtask is not directly model-checkable, the manager invokes a user interface to input:

- Selected reduction algorithms and their application order;
- Sub-properties of a complex xUML property;
- Boundaries and environment assumptions of a system component;
- Correspondence between sub-properties and components (or units of work);
- Class instances involved in a unit of work;
- Abstract state models and their corresponding concrete state models;

- Abstract data types and their mapping relations to concrete data types;
- Symmetries among class instances (or properties).

The inputs form a reduction conjecture. The manager applies the selected user-driven reduction algorithms in the user-defined order and generates subtasks. The manager then validates the reduction conjecture by invoking either the basic model checking process or a theorem prover. If the reduction conjecture is not valid, an error handling user interface is invoked to report the error and request a new reduction conjecture or termination of the model checking process.

If the current subtask is model-checkable, the manager invokes the basic model checking process to discharge the task. Several tasks can be discharged simultaneously if there is no dependency among them. If a subtask is checked to be false, the manager rolls the whole model checking process back to the reduction that generates the false task and invokes the error handling interface.

5.4 Framework Instantiation on Transaction Systems

Transaction systems such as banking systems and online sale systems play more and more important roles in the electronic infrastructure of our society. These systems are complex and require high reliability. Their designs follow similar patterns. Therefore, it is worthwhile to instantiate the integrated state space reduction framework for model checking xUML models of transaction systems.

5.4.1 Common Patterns of Transaction Systems

A transaction system executes transactions concurrently. A transaction consists of sequences of interactions among system components. Transactions may be of different types and transactions of the same type are often symmetric. The correctness of the system can be established by determining the correctness of each transaction it performs and the correctness of interactions among transactions.

Definition 5.4.1 *The model, M, of a transaction system, S, is the xUML model of S, which consists of a set of interacting class instances. A model, M', is a sub-model of M if M' consists of a subset of class instances of M.*

Definition 5.4.2 *A transaction type, T, of M is a message sequence template, which consists of sequences of message types defined in M. An instance of T is a transaction executed by M, whose message sequences follow T. A type, T', is a subtype of T if each sequence in T' is a sub-sequence of a sequence in T.*

Definition 5.4.3 *A transaction property, P, is a temporal logic predicate over all instances of a transaction type, T, or over an instance of T.*

Definition 5.4.4 *A model checking task is a tuple, $< M, T, P, A >$, where M is a model, T is a transaction type defined on M, P is a transaction property defined on T, and A is the set of assumed temporal properties defined on the environment of M. The environment of M is the aggregation of all inputs to M. A model checking task, $< M', T', P', A' >$, is a subtask of $< M, T, P, A >$ if M' is a sub-model of M, T' is a subtype of T, and P' is a temporal predicate that is defined on M' and derived from P through reductions such as decompositions, and A' is the union of*

A and a set of assumed properties on $M - M'$. Each assumed property in A or A' is a tuple of a temporal predicate and a model (or the environment) on which the predicate is defined.

Definition 5.4.5 *A model checking task, $< M, T, P, A >$, is directly model-checkable if it can be discharged by the basic model checking process using a reasonable amount of time and memory.*

5.4.2 Domain-Specific Reduction Algorithm

The domain-specific reduction algorithm for checking a task, $< \hat{M}, \hat{T}, \hat{P}, \hat{A} >$, on a transaction system is given in Figure 5.3. For simplicity, only the reduction aspect of the algorithm is covered in Figure 5.3. The algorithm constructs the reduction tree for $< \hat{M}, \hat{T}, \hat{P}, \hat{A} >$ on-the-fly. The root of the tree is $< \hat{M}, \hat{T}, \hat{P}, \hat{A} >$. Each non-root node in the tree is a subtask of its parent. The tree is expanded in a breadth first fashion. Every execution of the do loop either discharges a task at a leaf of the tree or expands the tree by reducing the task into its subtasks through symmetry reduction, decomposition, or case splitting. The expansion stops when all subtasks at the leaves of the tree are directly model-checkable.

5.4.3 Case Study: An Online Ticket Sale System

The xUML model of an online ticket sale system [64], M_0, is employed to illustrate the domain specific reduction algorithm for transaction systems. There are four classes in the system: Customer, Dispatcher, Agent, and Ticket Server. Both the Dispatcher class and the Ticket Server class have only one instance. The Agent class and the Customer class may have an arbitrary number of instances. The system

Enqueue($ToDo$, $< \hat{M}, \hat{T}, \hat{P}, \hat{A} >$); $Done$ ={ };
Do
  $< T, M, P, A >$ = Dequeue($ToDo$);
  If ($< T, M, P, A >$ is Directly Model-Checkable) Then
   Model check $< T, M, P, A >$; $Done = Done + \{< T, M, P, A >\}$; Continue;
  End;
  If (P is a property over all instances of T) Then
   Reduce P with Symmetry Reduction to P_1 where P_1 is on Instance 1 of T;
   Enqueue($ToDo$, $< T, M, P_1, A >$); Continue;
  End;
  If (M consists of instances from different classes) Then
   $Current$ = The first class that appears in T;
   Decompose M into M_1={All instances of $Current$} and M_2=$M - M_1$;
   Decompose T into T_1 performed by M_1 and T_2 performed by M_2;
   Decompose P into $P_1, \ldots, P_i$ on M_1 and $P_{i+1}, \ldots, P_m$ on M_2;
   $U_1 = \{P_1, \ldots, P_i\}$; $U_2 = \{P_{i+1}, \ldots, P_m\}$; $D_1 = \{\ \}$; $D_2 = \{\ \}$;
   While(!Empty(U_1) or !Empty(U_2))
     If (!Empty(U_1)) THEN
      P' = Remove-an-element(U_1); $A' = \{$Assumptions of P' on $M_2\}$
      Enqueue($ToDo$, $< T_1, M_1, P', A' >$); $D_1 = D_1 + \{P'\}$;
      $U_2 = U_2 + A' - D_2$;
     End;
     If (!Empty(U_2)) THEN
      P'' = Remove-an-element(U_2); $A'' = \{$Assumptions of P'' on $M_1\}$
      Enqueue($ToDo$, $< T_2, M_2, P'', A'' >$); $D_2 = D_2 + \{P''\}$;
       $U_1 = U_1 + A'' - D_1$;
     End;
   End;
  End;
  If (M consists only of all instances of a class, C) Then
   Reduce M with Case Splitting to M_1 where $M_1 = \{$Instance 1 of $C\}$;
   Enqueue($ToDo$, $< T, M_1, P, A >$); Continue;
  End;
Until (Empty($ToDo$));

Figure 5.3: Domain specific reduction algorithm for transaction systems

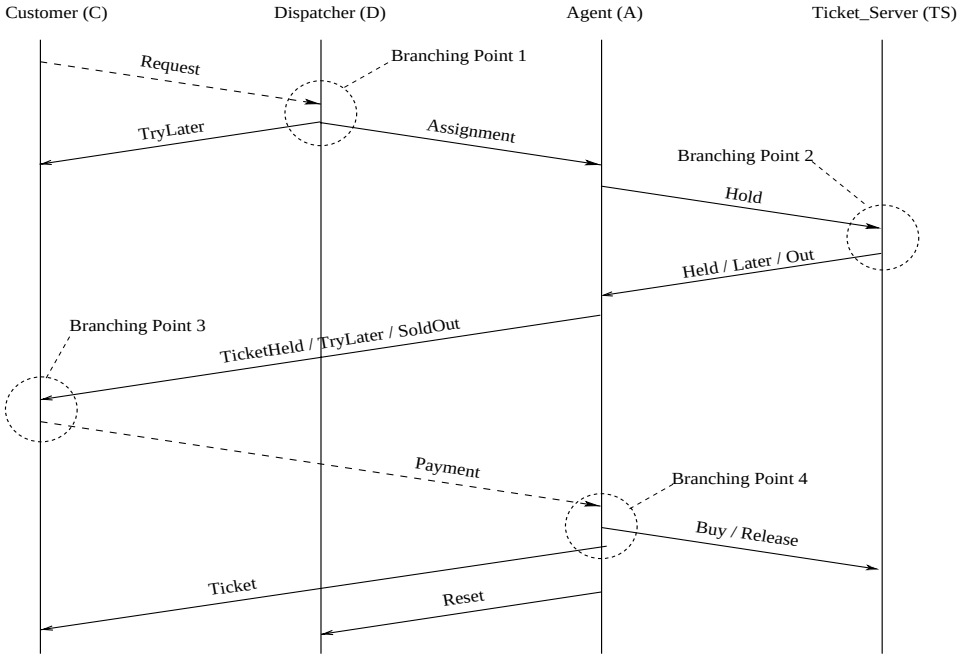

Figure 5.4: Message sequence diagram of ticketing transaction

processes ticketing transactions of the type, T_0, concurrently for many customers. The message sequence diagram of T_0 is shown in Figure 5.4. T_0 has four branching points where the decisions made affect the message sequences:

1. Upon processing a *request* message from a customer, the dispatcher assigns an idle agent to the customer if there is an idle agent; Otherwise, the dispatcher replies to the customer with a *TryLater* message;

2. Upon processing a *Hold* message from an agent, the ticket server replies to the agent with: A *Held* message if the number of tickets available is greater than the requested number; A *Later* message if the sum of tickets available or being held is greater than the requested number; An *Out* message otherwise;

3. Upon receiving a *TicketHeld* message from an agent, the customer may or may not reply to the agent with its payment;

4. If the valid payment from the customer is received before the agent times out, the agent sends a *Ticket* message to the customer and a *Buy* message to the ticket server; Otherwise, it sends a *Release* message to the ticket server.

A property which should hold on each transaction of the type, T_0, is that after a *request* message from a customer is processed by the dispatcher, eventually the system will send a *TicketHeld* message, or a *TryLater* message, or a *SoldOut* message back to the customer. The property is formulated as P_0 in Figure 5.5 using the xUML level property specification language introduced in Section 3.4. For simplicity, in Figure 5.5 some details are left out and i (or j, respectively) is used to index a general instance of the Customer class (or the Agent class).

Although the structure of the system is simple, the arbitrary number of customers and agents make directly model checking P_0 infeasible even for the most powerful model checkers. Therefore, the domain specific reduction algorithm is applied to reduce the model checking task, $< M_0, T_0, P_0, \Phi >$. The assumption set is empty since customers are also modeled as class instances in M_0. The sub-properties involved in the reduction process are defined in Figure 5.5. The sub-transactions and the sub-models involved in the process are shown in Figure 5.6. The reduction tree generated by the process is shown in Figure 5.7. Assumptions of a subtask are represented in Figure 5.7 by dashed arrows which lead to the subtasks that check the assumed properties on the corresponding sub-models. Reductions applied in the process are grouped into six general steps as follows:

P_0 : **After** Request(i) **Eventually** TicketHeld(i) or TryLater(i) or SoldOut(i)

P_1 : **After** Request(1)
Eventually TicketHeld(1) or TryLater(1) or SoldOut(1)

P_{21}: **After** Request(1) and **Forall** k { D.Agent_Free[k] = FALSE }
Eventually TryLater(1)
P_{22}: **After** Request(1) and **Exists** k { D.Agent_Free[k] = TRUE }
Eventually Assignment(j, 1) and A(j).$ = Idle
/* A(j).$ represents the current state of the class instance, A(j). */
P_{23}: **After** Assignment(j, 1) and A(j).$ = Idle
Eventually TicketHeld(1) or TryLater(1) or SoldOut(1)

P_{31}: **After** A(j).$ = Idle **Always** A(j).$ = Idle **UntilAfter** Assignment(j)
P_{32}: **After** Assignment(j) and A(j).$ = Idle **Eventually** Reset(j)
P_{33}: **After** Reset(j) **Eventually** A(j).$ = Idle

P_{41}: **After** A(1).$ = Idle **Always** A(1).$ = Idle **UntilAfter** Assignment(1)
P_{42}: **After** Assignment(1) and A(1).$ = Idle **Eventually** Reset(1)
P_{43}: **After** Reset(1) **Eventually** A(1).$ = Idle
P_{44}: **After** Assignment(1) and A(1).$ = Idle
Eventually TicketHeld(1) or TryLater(1) or SoldOut(1)

P_5 : **After** Hold(j) **Eventually** Held(j) or Later(j) or Out(j)

P_6 : **After** Hold(1) **Eventually** Held(1) or Later(1) or Out(1)

Figure 5.5: Original property and all intermediate sub-properties

Step 1: Symmetry Reduction P_0 is a temporal predicate over all transactions of the type T_0. Since customers are symmetric to each other, checking P_0 on M_0 is reduced to checking P_1 on M_0 where P_1 is a predicate only over the transaction that involves Customer 1.

Step 2: Compositional Reasoning T_0 is decomposed into three sub-transaction types, T_{11}, T_{12} and T_{13}. M_0 is decomposed into three sub-models, M_{11}, M_{12}, and M_{13}. Transactions of the types, T_{11}, T_{12} or T_{13}, are conducted by M_{11}, M_{12}, or M_{13}

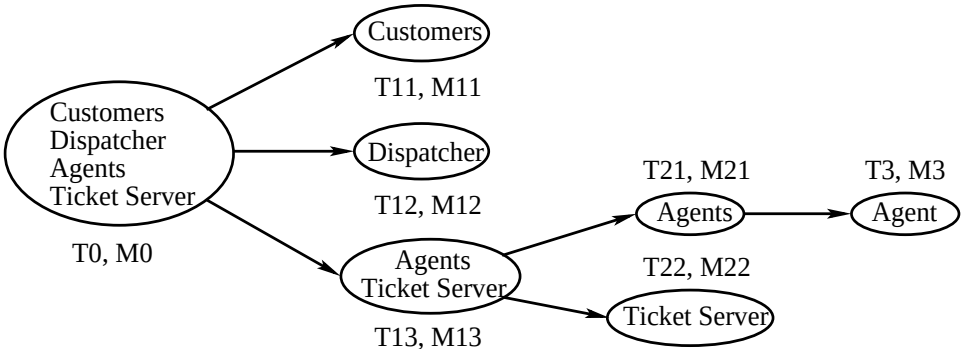

Figure 5.6: Decomposition relations among sub-models involved in reduction

respectively. P_1 is decomposed into three sub-properties: P_{21}, P_{22}, and P_{23}. P_{21} is directly model-checkable on M_{12} without any assumption on M_{11} or M_{13}. P_{22} is directly model-checkable on M_{12} by assuming that P_{31}, P_{32}, and P_{33} hold on M_{13}.

Step 3: Symmetry Reduction In M_{13}, agents are symmetric. P_{31}, P_{32}, P_{33}, and P_{23} have no assumption on M_{11} and M_{12}. Therefore, checking P_{31}, P_{32}, P_{33}, and P_{23} on M_{13} is reduced to checking P_{41}, P_{42}, P_{43}, and P_{44} on M_{13}.

Step 4: Compositional Reasoning T_{13} is further decomposed into two subtypes: T_{21} and T_{22}. Accordingly, M_{13} is decomposed into two sub-models: M_{21} and M_{22}. Transactions of the types T_{21} or T_{22} are conducted by M_{21} or M_{22} respectively. Checking P_{41}, P_{42}, P_{43}, and P_{44} on M_{13} is reduced to checking P_{41}, P_{42}, P_{43}, and P_{44} on M_{22} by assuming P_5 holds on M_{22}.

Step 5: Case Splitting In M_{21}, under the assumption P_5 on M_{22}, transactions of the type T_{21} and performed by agents are independent of each other. Therefore P_{41}, P_{42}, P_{43}, and P_{44} is instead checked over M_3 by assuming P_5 on M_{22}.

Step 6: Symmetry Reduction In M_{22}, transactions of the type, T_{22}, are symmetric. Therefore, checking P_5 on M_{22} is reduced to checking P_6 on M_{22}.

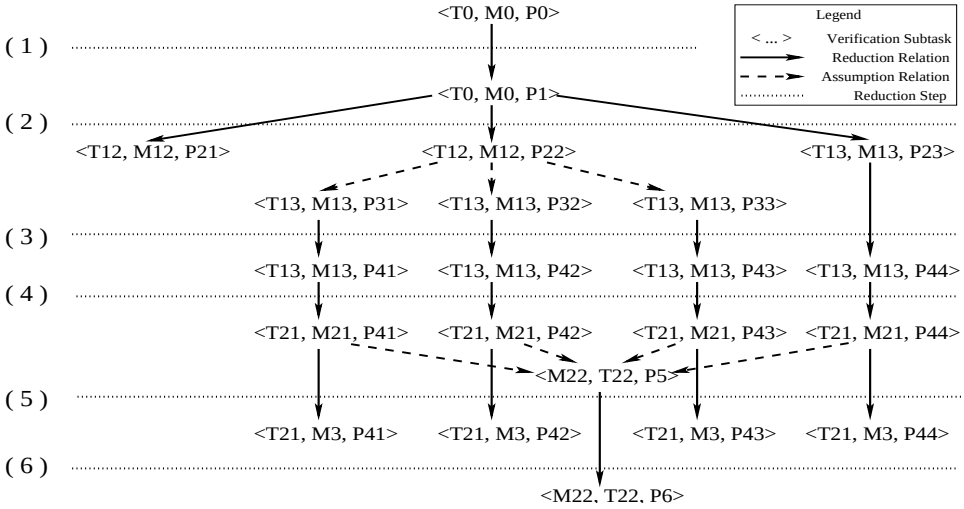

Figure 5.7: Reduction tree for verifying P_0 on online ticket sale system

5.5 Evaluation of Integrated State Space Reduction

Under the general framework, reduction algorithms applied in the user-driven reduction procedure recursively break a complex model checking task into subtasks that are directly model-checkable while reduction algorithms applied in the other two procedures facilitate directly model checking larger tasks. In this section, experiment results from the model checking study of the online ticket sale system are employed to evaluate the integrated application of these algorithms.

5.5.1 Evaluation of User-Driven Reduction Algorithms

Statistics from model checking Property P_0 in Figure 5.5 on the xUML model of the online ticket sale system are employed to demonstrate the effectiveness of the user-driven reduction algorithms. The memory and time usage for directly model checking P_0 is compared with the memory and time usage for checking the subtasks

generated by applying these reduction algorithms.

Directly model checking P_0 on the xUML model with two customer instances and two agent instances requires two separate model checking runs, one for each customer instance. With state partial order reduction (SPOR) and symbolic model checking (SMC) applied, each run takes 152.79 megabytes and 16273.7 seconds. The complexity of the xUML model increases rapidly as the number of customers increases. Directly model checking P_0 on the xUML model with 6 customer instances cannot be fulfilled. Therefore, directly model checking the xUML model with arbitrary number of customers is not feasible.

The memory and time usage for model checking each subtask from the reduction tree in Figure 5.7 is shown in Table 5.1. It can be observed that the memory

Criteria	P_{21}	P_{22}	P_{41}	P_{42}	P_{43}	P_{44}	P_6
Memory	0.30M	0.95M	0.28M	0.29M	0.28M	0.29M	0.35M
Time	0.02S	1.81S	0.01S	0.04S	0.01S	0.04S	0.63S

Table 5.1: Time and memory usage of subtasks in verifying P_0

and time usage for each subtask is substantially lower than that for directly model checking P_0 on the xUML model with two customers. The model checking result from the reduction process can be scaled up to xUML models with arbitrary number of customer and agent instances by further applying non-deterministic abstraction and symmetry reduction. The complexity of symmetric property reduction is not shown due to an unfinished feature of our reduction system. However, the complexity is theoretically bounded by the complexity of the static structure of an xUML model because the reduction only checks the static structure of the model instead of exploring the full state space of the model.

5.5.2 Evaluation of SPOR, SMC, and Their Combined Application

Being able to directly discharge larger model checking tasks reduces user interaction and makes the integrated state space reduction more automatic. Currently, to scale up directly model-checkable tasks, SPOR is applied in the xUML-to-S/R translation and SMC is applied in the S/R level model checking. To demonstrate the reduction

SPOR	SMC	Memory Usage	Time Usage
Off	Off	167.072M	193748S
On	Off	16.0604M	10476.5S
Off	On	142.746M	471.32S
On	On	102.527M	280.1S

Table 5.2: Model checking memory and time usage comparison

ability of SPOR and SMC, Property P_{21} in Figure 5.5 is directly checked on the whole model under the four possible on/off combinations of SPOR and SMC. The model checking complexities under the four combinations are compared in Table 5.2. It can be observed that both SPOR and SMC lead to significant reduction on the model checking complexity. SPOR offers a better memory usage while SMC offers a better time usage. Their combined application achieves the best time usage with a medium memory usage.

5.6 Summary

This chapter defines and describes a general framework for integrated state space reduction in model checking executable software system designs. The framework is presented for system designs modeled in xUML, but is readily applicable to other representations. Partially implemented automaton support for the framework is

discussed. The framework is illustrated by its instantiation for distributed transaction systems and is evaluated by applying the instantiation in model checking an online ticket sale system. The dimension of the software system designs that are model-checkable is found to be substantially extended.

In essence, the integrated state space reduction framework conducts a systematic *top-down* decomposition of a complex software system into directly model-checkable components by exploring domain-specific knowledges. The framework explores the compositional structures of software systems. In CBD, a system is developed through *bottom-up* component composition and the compositional structures of the system and its recursive components are intentionally introduced. In Chapter 6, we discuss how to integrate model checking into CBD by exploring the compositional structures introduced by CBD.

Chapter 6

Verified Systems by Composition from Verified Components

6.1 Motivation and Overview

Component-Based Development (CBD) [61], developing software systems through composition of components, is one of the most important technical initiatives in software engineering. Model checking provides exhaustive state space coverage for the systems being checked and is particularly effective at detecting coordination errors which frequently result from component compositions and are notoriously difficult to detect. However, model checking often cannot handle large-scale software systems due to state space explosions. Model checking and CBD are synergistic. Model checking can potentially enable effective development of more reliable component-

based software systems. CBD introduces compositional structures, clean component interfaces, and standard composition rules to the systems being built, which may reduce the state spaces that model checkers have to handle.

This chapter defines, discusses, and illustrates an approach to integration of model checking into the CBD of software systems, which contributes to solution of the following fundamental problems in CBD and model checking:

- Developing components which can be reused with certainty that their behaviors will meet their specifications in a proper composition;
- Identifying proper components for a composition;
- Establishing that a component composed from "correct" components will meet its specifications;
- Alleviating the state space explosion problem.

This approach can be summarized as follows:

- As a software component is built, temporal properties of the component are specified, verified, and then packaged with the component.
- Selecting a component for reuse considers not only its functionality but also its temporal properties.
- Verification of properties of a composed component reuses verified properties of its sub-components and is based on compositional reasoning [53, 1, 3, 46, 4, 19].

A general component model that enables component verification is defined. This model provides a framework for representing components and their properties

and for composing components. In this model, a property of a component is defined with assumptions on the environment of the component. The property is verified on the component under these assumptions. When the component is reused in the composition of a larger component, the verified property is *enabled* if the environment assumptions made in its verification hold on other components in the composition and/or the environment of the composed component. (The formal definition of an enabled property is given in Section 6.3.5).

The general component model can be instantiated on existing computation models upon which syntax and semantics for components, properties, and component compositions are precisely defined. These computation models also provide semantics for component execution and interaction. We demonstrate the general component model with its instantiation on an Asynchronous Interleaving Message-passing (AIM) computation model. In this instantiation, executable representations of components are specified in xUML [47], an executable dialect of UML, whose semantics conform to the AIM model. This instantiation is of interest because the AIM model captures the essential nature of a broad range of concurrent software systems.

Components are categorized as *primitive* components (components that are built from "scratch" and not composed from other components) or as *composed* components. A property of a primitive component is verified by directly model checking an executable representation of the component, for instance, checking the executable design model (specified in xUML) of the component using methods established in Chapters 3, 4, and 5. A property of a composed component, instead of being model checked on the executable representation of the component, is checked

on an *abstraction* of the component. The abstraction is composed of simple automata corresponding to environment assumptions of the composed component and verified properties of its sub-components. A verified sub-component property is included in the abstraction if it is enabled in the composition, related to the property to be checked by cone-of-influence analysis [17], and not involved in invalid circular dependencies [46] among sub-component properties. If the abstraction is still too complex to be checked directly, compositional reasoning is applied to decompose the abstraction. If the abstraction is too abstract to enable the verification of the desired property, it is refined as follows: decomposing the property into a set of properties of the sub-components, verifying these properties on the sub-components, and then including these verified properties into the abstraction. Algorithms for abstraction construction and refinement are based on compositional reasoning.

Our approach is most suitable for application to a product line of software systems that are built from a growing set of software components. We have identified two major application domains: product lines of software systems based on a specific hardware/software architecture, such as the TinyOS [30] run-time system, and product lines of distributed large-scale software systems based on component platforms such as CORBA and DCOM.

The rest of this chapter is organized as follows. Section 6.2 defines the general component model for component verification and instantiates it on the AIM computation model. Section 6.3 elaborates on how to verify properties of components, either primitive or composed. Section 6.4 illustrates our approach with a case study on TinyOS. Section 6.5 analyzes the effectiveness of our approach in the context of the case study. Section 6.6 presents the related work. Section 6.7 summarizes.

6.2 Component Model for Verification

In this section, we first define the general component model, which is a template that can be instantiated on a computation model to enhance it with components, properties, and component compositions which enable component verification. After that, we instantiate the general component model on the AIM computation model.

6.2.1 General Component Model

Component

A component, C, is a four-tuple, $(E,\ I,\ V,\ P)$, where

- E is an executable representation of C.
- I is an interface through which C interacts with other components, for instance, a messaging interface or a procedural interface.
- V is a set of variables defined in E and referenced by the properties defined in P.
- P is a set of temporal properties that are defined on I and V, and have been verified on E. A temporal property is denoted by a pair, $(p,\ A(p))$, where p is a temporal formula defined on I and V, and $A(p)$ is a set of temporal formulas defined on I and V. The property, p, holds on C if the temporal formulas in $A(p)$ hold on the *environment* of C (see the next paragraph for the definition of the environment of a component). P is extended incrementally by including properties that are newly verified. A property is included in P only when it is verified.

A system is a component. The environment with which the system interacts is also modeled as a set of components. The set of components with which a component interacts is referred to as the environment of the component. The environment of a component varies as the component is reused in different compositions. Given a component and its property, $(p,\ A)$, the temporal formulas in A are referred to as the environment assumptions of the component for enabling p.

Component Composition

A component, $C = (E,\ I,\ V,\ P)$, can be composed from a set of simpler components, $(E_0,\ I_0,\ V_0,\ P_0)$, ..., $(E_{n-1},\ I_{n-1},\ V_{n-1},\ P_{n-1})$, as follows:

- E is constructed from E_0, ..., E_{n-1} by connecting E_0, ..., E_{n-1} through their interfaces.
- I is derived from I_0, ..., I_{n-1}: An operation in I_i, $0 \leq i < n$, is included in I if and only if it is used when C interacts with other components.
- V is a subset of $\bigcup_{i=0}^{n-1} V_i$. A variable in $\bigcup_{i=0}^{n-1} V_i$ is included in V if and only if the variable is referenced by the properties defined in P.
- P is a set of temporal properties defined on I and V, and verified on E. Properties in P are verified on E by utilizing the properties in P_0, ..., P_{n-1}.

6.2.2 Instantiation of General Component Model on AIM Computation Model

The AIM computation model is basically the AIM semantics, which is informally described in Section 4.3.1 and formalize in Section 4.3.2, and the simple syntax used

in Section 4.3.2 to present the AIM semantics. This section discusses the instantiation of the general component model on the AIM computation model with the focus on how to derive the executable specification and the interface of a composed component from its sub-components.

Component

A component, C, is a four-tuple, $(E,\ I,\ V,\ P)$, where

- E is an executable representation of C with syntax and semantics conforming to the AIM computation model. E can be either an AIM specification that consists of a set of interacting AIM processes, or an implementation of the AIM specification in a programming language.
- I is the messaging interface through which C interacts with other components and is a pair, $(R,\ S)$, where R (or S, respectively) is a set of input (or output) message types whose instances may be input (or output) by C, more precisely by processes in C, when C interacts with other components.
- V and P inherit their definitions from the general component model and reference semantic entities in the instantiations of E and I on the AIM computation model.

Component Composition

A component, $C = (E,\ I,\ V,\ P)$, can be composed from a set of simpler components, $(E_0,\ I_0,\ V_0,\ P_0)$, ..., $(E_{n-1},\ I_{n-1},\ V_{n-1},\ P_{n-1})$, as follows:

- E is constructed from E_0, ..., E_{n-1}, by mapping output message types in

S_0, ..., S_{n-1} to input message types in R_0, ..., R_{n-1}. If there is a mapping defined between an output message type, s, in S_i, $0 \leq i < n$, and an input message type, r, in R_j, $0 \leq j < n$, the following steps are executed:

  - A conformance check is performed on the parameter lists of s and r;
  - All occurrences of s in E_i are replaced by r, except the occurrences where messages of the type, s, are output to components outside C;
  - If s is mapped to more than one input message type, a messaging statement that outputs s is replicated for each input message type.

- $I = (R,\ S)$ is derived from $I_i = (R_i,\ S_i)$, $0 \leq i < n$. R (or S, respectively) is a subset of $\bigcup_{i=0}^{n-1} R_i$ (or $\bigcup_{i=0}^{n-1} S_i$). A message type in $\bigcup_{i=0}^{n-1} R_i$ (or $\bigcup_{i=0}^{n-1} S_i$) is included in R (or S) if and only if messages of that type may be input (or output) by C when C interacts with other components.

- V is derived by following the corresponding rule in the general component model.

- Formulation of the properties in P and verification of these properties by utilizing the properties in P_0, ..., P_{n-1} are discussed in Section 6.3 in detail.

Component Execution and Interaction

The execution semantics of components are defined recursively (assuming bounded recursion). When a component executes, if the component has no sub-components, then at any given moment exactly one AIM process in the component executes; if the component has sub-components, then at any given moment exactly one sub-component executes.

Components interact with each other through message-passing. A component can only input (or output, respectively) messages of the types listed in its input (or output) messaging interface. Messages input (or output) by a component are consumed (or generated) by AIM processes in the component or its recursively nested sub-components.

6.3 Verification of Components

This section discusses verification of components under the instantiation of the general component model on the AIM computation model. First, we introduce how a closed AIM system is verified. Then, we discuss how component properties are formulated. Finally, we differentiate components into two categories, *primitive* and *composed*, and present procedures for verifying components of the two categories respectively.

6.3.1 Background: Verification of a Closed AIM System

There are many software design specification languages whose semantics conform to the AIM model, such as xUML [47] and SDL [35]. A closed AIM system can be specified in these languages. In Chapter 3, we have presented a translation-based approach to model checking executable software system designs in xUML, which is supported by the the ObjectCheck toolkit. This approach requires that a system design to be checked specify a closed system. A system is made closed by modeling its environment as part of the system.

This approach suffers from the state space explosion problem. To verify large-scale software system designs, we have extended the approach with an inte-

grated state space reduction framework, presented in Chapter 5. This framework features a top-down application of compositional reasoning, where model checking a property on a system is accomplished by decomposing the system into modules, checking module properties locally on the modules, and deriving the system property from the module properties. We applied compositional reasoning in model checking xUML specifications by following the Translation-Based Compositional Reasoning approach, presented in Chapter 4, where compositional reasoning rules are established in the semantics of software systems, but are proved and implemented based on translation of software systems to formal representations for which compositional reasoning rules have already been established, proved, and implemented.

6.3.2 Formulation of Component Properties

After the AIM specification and the messaging interfaces of a component are constructed, properties of the component can then be formulated. Properties are mainly derived from functional specifications of the component such as input and output relationships through domain analysis. However, it is not required that all properties of the component be packaged initially. Additional properties may be introduced incrementally as the component is reused in composing other components. Verification of a property on a composed component may require top-down application of compositional reasoning to decompose the property into a set of properties on its sub-components. The sub-component properties are then verified on the sub-components and packaged with the sub-components for future reuse. This top-down application of compositional reasoning requires that system/component developers manually guide the property decomposition, however, it enables verification of large-

scale components that cannot otherwise be verified.

Since our approach targets a product line of software systems constructed from a set of components, we assume that the AIM specifications of the components are available to system/component developers, which facilitates incremental introduction and verification of component properties. The set of properties of a component is expected to become quite stable after a few reuses.

6.3.3 Formulation of Environment Assumptions

Our approach requires that a property of a component be specified with its assumptions on the environment of the component. We have investigated both automatic generation and manual formulation of environment assumptions. Given a component and a property, an assumption that enables the property on the component can be automatically generated by taking the complement of the product of the property and its cone-of-influence on the component. (Various optimizations are possible.) Construction of the complement, the product, and the cone-of-influence is supported by COSPAN. The assumption generated is the weakest assumption that enables the property on the component, however, it is usually a complex and non-intuitive assumption which is difficult to check on other components composed with the component in a composition. A desired set of assumptions for the property is a set of simple and intuitive assumptions formulated on the interface of the component. Each of these assumptions is weaker than the automatically generated assumption, however, the conjunction of these assumptions is stronger than the automatically generated assumption. These assumptions are often easier to check on other components. Domain-specific knowledge of system/component developers

is expected to facilitate the formulation of such a set of assumptions. Therefore, currently in our approach environment assumptions are formulated manually. Investigation of heuristics that can reduce or decompose the automatically generated assumption by utilizing domain-specific knowledge is in progress.

6.3.4 Verification of Primitive Components

A primitive component often has limited functionality. As a result, the state space of a primitive component is often of modest size and suitable for direct application of model checking. The approach in Section 6.3.1 is employed to verify a primitive component. However, the AIM specification of a primitive component often does not specify a closed system and the approach cannot be readily applied. Therefore, we construct a closed system from the AIM specification and the environment assumptions of the component.

Given a primitive component, $C = (E,\ I,\ V,\ P)$, and a property, $(p,\ A(p))$, specified on I and V, in order to check whether p holds on E assuming that assumptions in $A(p)$ hold on the environment of C, the following steps are executed:

1. Create an AIM process, ENV, whose input message types are the same as the output message types defined in I and whose state model outputs messages of the input message types defined in I;

2. Build an AIM system from ENV and the AIM processes in E and translate the system into S/R;

3. Free all variables of the automata corresponding to ENV in the S/R model obtained in Step 2 so that these variables may obtain any value in their do-

mains non-deterministically; (Discussions on freeing variables in an S/R model can be found in [28, 37].)

4. Translate assumptions in $A(p)$ to S/R automata and compose them with the S/R model obtained in Step 3 so that the free variables introduced in Step 3 are now constrained by the assumptions in $A(p)$;

5. Translate p to an S/R property and check the S/R property on the S/R model gotten in Step 4.

These steps construct a closed system by using the ENV process as a translation stub and replacing ENV with the assumptions in $A(p)$ in the resulting S/R model, and then verify p on the closed system. Construction of the closed system is simplified by the fact that in S/R, models, properties, and assumptions are all specified as automata that are of the same form and can be trivially composed.

6.3.5 Verification of Composed Components

In this section, we present a method for verification of a property, $(p,\ A(p))$, on a composed component, $C = (E,\ I,\ V,\ P)$, where C is composed from $C_0 = (E_0,\ I_0,\ V_0,\ P_0)$ and $C_1 = (E_1,\ I_1,\ V_1,\ P_1)$. This method reuses the properties that have been verified on the sub-components, C_0 and C_1. It can be readily extended to the case that C is composed from $C_0, \ldots, C_{n-1}$.

Component Abstraction Construction

Since the AIM specification of a composed component often has a large state space that hinders direct application of model checking, we construct an abstraction of

the component based on the composition, the environment assumptions of the component, the messaging interfaces of the sub-components, and the verified properties of the sub-components.

Before discussing how to construct the abstraction, we first elaborate on the concept of *enabled property*. A property of a component is defined with assumptions on the environment of the component. The property is verified on the component under these assumptions. When the component is reused in the composition of a larger component, the property is *enabled* if the environment assumptions made in its verification hold on other components in the composition and/or the environment of the composed component. We now formally define an *enabled property*.

Definition 6.3.1 *Enabled Property* A property $(p_i,\ A(p_i))$ of C_i, where $i \in \{0,\ 1\}$ and $(p_i,\ A(p_i)) \in P_i$, is enabled in the composition of C_0 and C_1 if and only if either $A(p_i)$ is empty or for each q, $q \in A(p_i)$, q is implied by the assumptions in $A(p)$ and the properties in P_{1-i} that are enabled in the composition.

The function in Figure 6.1 can be used to determine whether a property $(p_i,\ A(p_i))$

```
boolean function enabled ( (p_i, A(p_i)) ) begin
  while ( !empty( A(p_i) ) ) do
    q = remove-an-element ( A(p_i) ); P'={ };
    foreach ((p', A(p')) ∈ cone(A(P) ∪ P_{1-i}, q) ∩ P_{1-i})
      if ( enabled ( (p', A(p')) ) then
        P'=P' ∪ {(p', A(p'))};
      endif;
    endfor;
    if (q is implied by cone(A(p) ∪ P', q)) then continue;
    else return false;
    endif;
  endwhile;
  return true;
end;
```

Figure 6.1: The "enabled" function

of C_i, $i \in \{0,\ 1\}$, is enabled in the composition of C_0 and C_1 assuming that the assumptions in $A(p)$ hold on the environment of the composition. For each assumption q, $q \in A(p_i)$, the function first identifies the set P' of properties that are in P_{1-i}, related to q according to cone-of-influence analysis, and enabled. (In Figure 6.1, $cone(A(p) \cup P_{1-i},\ q)$ denotes the cone-of-influence of q on $A(p) \cup P_{1-i}$. $cone(A(p) \cup P_{1-i},\ q)$ includes assumptions in $A(p)$ and properties in P_{1-i}, which reference the semantics entities in C that influence the semantics entities referenced by q.) If q is implied by $cone(A(p) \cup P',\ q)$, then the function continues with the next assumption in $A(p_i)$; otherwise, the function returns false. The implication can be decided by either matching q to an element of $cone(A(p) \cup P',\ q)$, or model checking q on the product of the elements of $cone(A(p) \cup P',\ q)$. If a property, $(p_i,\ A(p_i))$, of C_i is not currently enabled in the composition of C_0 and C_1, it does not indicate that p_i does not hold on C_i under the composition. $(p_i,\ A(p_i))$ can become enabled when all assumptions in $A(p_i)$ become enabled, which may require checking additional properties of C_0 and C_1.

The abstraction of the composed component, C, upon which the property, p, is to be verified, is derived as follows:

- Realize the output message interfaces of C_0 (or C_1, respectively) in the context of C by replacing the output message types of C_0 (or C_1) with the corresponding input message types of C_1 (or C_0) according to the mappings among the output message types of C_0 (or C_1) and the input message types of C_1 (or C_0);

- Create an AIM system, SYS, which consists of three stub AIM processes, CP_0, CP_1, and ENV, where: (i) CP_0 (or CP_1, respectively) is corresponding to C_0

(or C_1), whose variables have the same names and domains as the variables in V_0 (or V_1), whose input message types are the same as the input message types of C_0 (or C_1), and whose state model outputs messages of the output message types of C_0 (or C_1); (ii) ENV is corresponding to the environment of C, whose input message types are the same as the output message types of C and whose state model outputs messages of the input message types of C;

- Run the "enabled" function in Figure 6.1 on each property in P_0 and P_1, and include the property into SYS if the function returns true;
- Include the assumptions in $A(p)$ into SYS;
- Run the cone-of-influence analysis on SYS to exclude properties and assumptions not related to p.

There may exist circular dependencies among the sub-component properties. Circular dependencies among the sub-component properties may be invalid, i.e., may lead to circular reasoning. Suppose we have verified that a property, P, holds on C_0 assuming that a property, Q, holds on C_1 and vice versa. We cannot conclude that P and Q hold on C unless we can show that the circular dependency between P and Q is valid, i.e., circular reasoning can be avoided. Circular reasoning can be avoided using the following methods (but not limited to these methods):

- avoid using an assumption that creates a dependency cycle;
- use temporal induction [46] proposed by McMillan; or
- use the composition reasoning rule [4] proposed by Amla, et al.

Circular reasoning avoidance can be readily included in the "enabled" function.

Verification of Component Abstraction

Instead of checking the property, p, on the AIM specification of C, we check p on the abstraction, SYS:

- Translate SYS into S/R;
- Free all variables of the automata corresponding to CP_0, CP_1, and ENV in the resulting S/R model; (These free variables are now constrained by properties from P_0 and P_1 and assumptions from $A(p)$.)
- Translate p into S/R and check the S/R query corresponding to p on the S/R model corresponding to SYS;
- Include $(p,\ A(p))$ in P if p holds on SYS; otherwise, refine SYS as discussed in Section 6.3.5.

The complexity of model checking p on the abstraction, SYS, is often much lower than the complexity of directly checking p on the AIM specification of C (see Section 6.5.2).

Refinement of Component Abstraction

If p does not hold on the abstraction, SYS, then either p does not hold on C assuming assumptions in $A(p)$ hold on the environment of C, or SYS is too abstract. It is often possible to differentiate the two cases by analyzing the error traces generated by the model checker. If p does not hold on C under the assumptions in $A(p)$, then either more assumptions have to be added to $A(p)$ or C has to be re-composed. If SYS is too abstract, it must be refined.

The abstraction can be refined by including additional properties of C_0 and C_1. These properties are either properties that are newly introduced, but have not been verified, or properties that have been verified, but are not currently enabled in the composition. If a property to be included has not been verified, it is first verified. If a property to be included has been verified, but is not currently enabled, the procedure in Figure 6.2 is applied to enable the property. The procedure enables

```
boolean procedure enable ( (p_i, A(p_i)) ) begin
  while ( !empty ( A(p_i) ) ) do
    q = remove-an-element ( A(p_i) ); P' = { };
    foreach ((p', A(p')) ∈ cone(A(p) ∪ P_{1-i}, q) ∩ P_{1-i})
      if (enabled ((p', A(p')))) then P' = P' ∪ {(p', A(p'))};
      elseif ( enable ((p', A(p'))) ) then
        P' = P' ∪ {(p', A(p'))};
      endif;
    endfor;
    if (q is implied by cone(A(p) ∪ P', q) ) then continue;
    elseif ( q is expected to hold on C_{1-i} ) then
      A' = { assumptions of q };
      if ( !verify ( (q, A'), 1 − i ) ) then return false; endif;
      return enable ( (q, A') );
    else return false;
    endif;
  endwhile;
return true;
end;
```

Figure 6.2: The "enable" procedure

a property, $(p_i,\ A(p_i))$, of C_i by enabling all its assumptions. For each assumption, q, the procedure first attempts to enable the properties that are in $cone(A(p) \cup P_{1-i},\ q) \cap P_{1-i}$ and not enabled, by calling itself recursively. After the *foreach* loop, P' contains all the properties in $cone(A(p) \cup P_{1-i},\ q) \cap P_{1-i}$ that have been enabled. If q is implied by $cone(A(p) \cup P',\ q)$, the procedure continues with the next assumption; otherwise, if q is expected to hold on C_{1-i} [1], a set of assumptions, A', of

[1]If q is expected to hold on the conjunction of C_{1-i} and $A(p)$, it is first decomposed into sub-assumptions on C_{1-i} and $A(p)$ respectively, which is guided by system/component developers.

q is introduced and $(q,\ A')$ is verified on C_{1-i}. If $(q,\ A')$ is successfully verified, the "enable" procedure is called on $(q,\ A')$ recursively. The "enable" procedure returns false if a call to the "verify" procedure returns false or if q is neither an assumption of the composed component, C, on its environment nor q is a property that is expected to hold on C_{1-i}. Circular dependencies among properties, introduced by the refinement, must be validated as discussed in Section 6.3.5.

6.4 Case Study: Verification of TinyOS Components

We have applied the approach to integration of model checking into CBD to improve reliability of instances of TinyOS [30]. We now illustrate this approach with a case study on TinyOS. TinyOS is a component-based run-time system designed to provide support for deeply embedded systems which require concurrency-intensive operations while constrained by minimal hardware resources. Hardware constraints of deeply embedded systems prohibit loading all TinyOS modules into a single instance and different requirements of these systems require different configurations of TinyOS modules, which makes CBD an appropriate development approach for TinyOS. TinyOS instances are usually loaded to a large number of deeply embedded systems such as networked sensors, which makes correction of software bugs very expensive. Locks and monitors, which are often used to safeguard concurrent operations, are not used in TinyOS due to their computational expenses and the hardware constraints of TinyOS. This combination of complexity and the requirement for high reliability justifies the application of our approach to improve reliability of instances of TinyOS.

6.4.1 Sensor Component

We sketch how primitive components are specified and verified with the Sensor component. We first introduce the (E, I, V, P) specification of the Sensor component. The executable representation, E, of the Sensor component is specified in xUML. The communication diagram of the Sensor component is shown in Figure 6.3. (Space

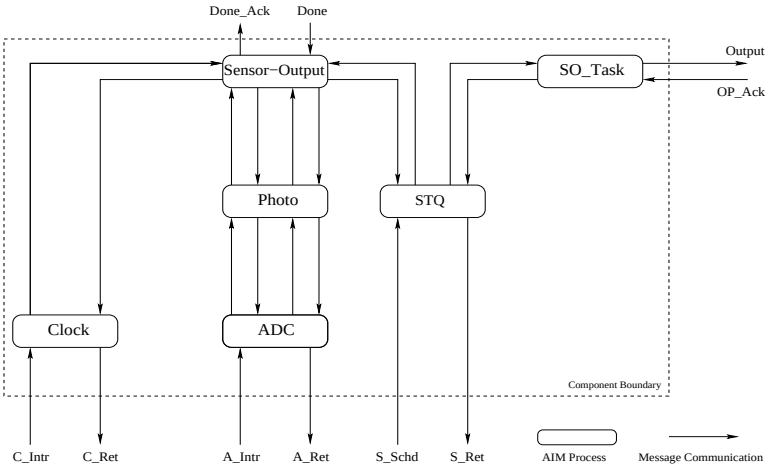

Figure 6.3: Sensor component

limitations prohibit showing all xUML diagrams of E.) The component consists of six AIM processes that interact with each other and the environment of the component through messages. The messaging interface, I, of the component is as follows:

- R={*C_Intr, A_Intr, S_Schd, OP_Ack, Done*};
- S={*C_Ret, A_Ret, S_Ret, Output, Done_Ack*}.

Message types in R are defined in the AIM processes of the Sensor component and the message types in S are to be realized when the component is composed with other components. *C_Intr*, *A_Intr*, and *S_Schd* are the hardware interrupts

the Sensor component must handle while *C_Ret*, *A_Ret*, and *S_Ret* are the corresponding replies. The Sensor component outputs Sensor readings as messages of the type, *Output*. The properties to be checked on the Sensor component are listed in Figure 6.4 with their assumptions. (In Figure 6.4, the "+" operator denotes a

Properties:
Repeatedly (Output);

After (Output) **Never** (Output) **UntilAfter** (OP_Ack);

After (Done) **Eventually** (Done_Ack);
Never (Done_Ack) **UntilAfter** (Done);
After (Done_Ack) **Never** (Done_Ack) **UntilAfter**(Done);

Assumptions:
After (Output) **Eventually** (OP_Ack);
Never (OP_Ack) **UntilAfter** (Output);
After (OP_Ack) **Never** (OP_Ack) **UntilAfter** (Output);

After (Done) **Never** (Done) **UntilAfter** (Done_Ack);

Repeatedly (C_Intr);
After (C_Intr) **Never** (C_Intr + A_Intr + S_Schd) **UntilAfter** (C_Ret);

After (ADC.Pending) **Eventually** (A_Intr);
After (A_Intr) **Never** (C_Intr + A_Intr + S_Schd) **UntilAfter** (A_Ret);

After (STQ.Empty = FALSE) **Eventually** (S_Schd);
After (S_Schd) **Never** (C_Intr + A_Intr + S_Schd) **UntilAfter** (S_Ret);

Figure 6.4: Properties of Sensor component

logical OR. Detailed discussions of the property specification language are given in Section 3.4.) These properties assert that the component repeatedly outputs sensor readings and correctly handles the signal-and-reply relationship between *Output* and *OP_Ack* and between *Done* and *Done_Ack* assuming that the assumptions hold on its environment. The set, V, consists of two variables, *ADC.Pending* and *STQ.Empty*, referenced by the properties and the assumptions listed in Figure 6.4.

The Sensor component has a state space of modest size. The properties listed in Figure 6.4 were successfully verified on the component by following the steps in Section 3.4 and were included into P for future reuse.

6.4.2 Network Component

The communication diagram of the Network component is shown in Figure 6.5. The

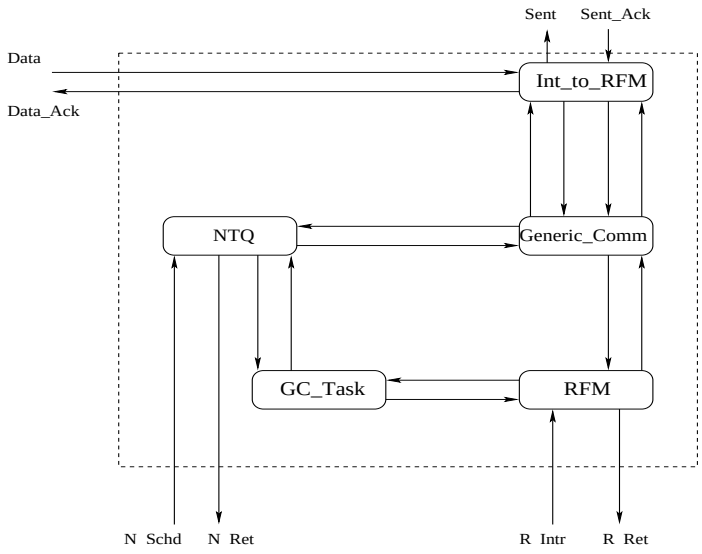

Figure 6.5: Network component

messaging interface, I, of the Network component is as follows:

- R={*N_Schd, R_Intr, Sent_Ack, Data*};
- S={*N_Ret, R_Ret, Sent, Data_Ack*}.

The properties that have been verified on the Network component are listed in Figure 6.6 with their assumptions. These properties assert that the Network component transmits on the physical network repeatedly if it receives inputs repeatedly, and it

Properties:
IfRepeatedly (Data) **Repeatedly** (RFM.Pending);
IfRepeatedly (Data) **Repeatedly** (Not RFM.Pending);

After (Data) **Eventually**(Data_Ack);
Never (Data_Ack) **UntilAfter** (Data);
After (Data_Ack) **Never** (Data_Ack) **UntilAfter** (Data);

After (Sent) **Never** (Sent) **UntilAfter** (Sent_Ack);

Assumptions:
After (Data) **Never** (Data) **UntilAfter** (Data_Ack);

After (Sent) **Eventually** (Sent_Ack);
Never (Sent_Ack) **UntilAfter** (Sent);
After (Sent_Ack) **Never** (Sent_Ack) **UntilAfter** (Sent);

After (NTQ.Empty = FALSE) **Eventually** (N_Schd);
After (N_Schd) **Never** (N_Schd + R_Intr) **UntilAfter**(N_Ret);

After (RFM.Pending) **Eventually** (R_Intr);
After (R_Intr) **Never** (N_Schd + R_Intr) **UntilAfter** (R_Ret);

Figure 6.6: Properties of Network component

correctly handles the signal-and-reply relationship between *Data* and *Data_Ack* and between *Sent* and *Sent_Ack*. The set, V, of the Network component consists of two variables, *RFM.Pending* and *NTQ.Empty*.

6.4.3 Sensor-to-Network Component

This section introduces how an instance of TinyOS, the Sensor-to-Network component, is composed from the Sensor component and the Network component, and discusses how properties of the composed component are verified by utilizing the properties that have been verified on its sub-components.

The executable representation, E, of the Sensor-to-Network component is composed from the executable representations of the Sensor component and the Network component. The abstracted communication diagram of the Sensor-to-Network

component is shown in Figure 6.7, where an annotation of the form of "input mes-

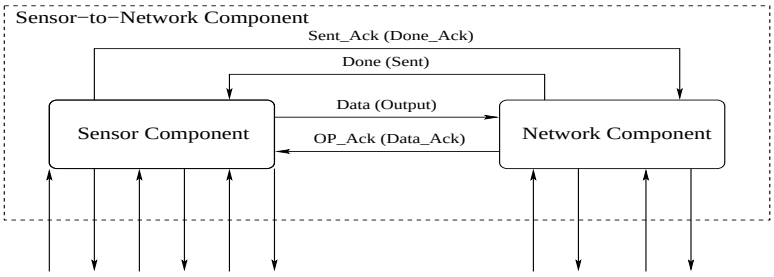

Figure 6.7: Sensor-to-Network component

sage type (output message type)" denotes the mapping of an output message type of a component to an input message type of the other component. The messaging interface, I, of the Sensor-to-Network component is as follows:

- R={*C_Intr, A_Intr, S_Schd, N_Schd, R_Intr*};
- S={*C_Ret, A_Ret, S_Ret, N_Ret, R_Ret*}.

The properties to be checked on the Sensor-to-Network component are listed in Figure 6.8 with their assumptions. These properties assert that the Sensor-to-Network component repeatedly transmits on the physical network if the assumptions hold on its environment. We refer to these properties together as the "repeated transmission" property hereafter. The set, V, of the Sensor-to-Network component consists of four variables, *ADC.Pending*, *STQ.Empty*, *RFM.Pending*, and *NTQ.Empty*.

In order to check the "repeated transmission" property on the Sensor-to-Network component, we constructed an abstraction of the component following the steps given in Section 6.3.5:

- Replace the output message types of the Sensor (or Network, respectively)

Properties:
Repeatedly (RFM.Pending);
Repeatedly (Not RFM.Pending);

Assumptions:
Repeatedly (C_Intr);
After (C_Intr) **Never** (C_Intr+A_Intr+S_Schd+N_Schd+R_Intr) **UntilAfter** (C_Ret);

After (ADC.Pending) **Eventually** (A_Intr);
After (A_Intr) **Never** (C_Intr+A_Intr+S_Schd+N_Schd+R_Intr) **UntilAfter** (A_Ret);

After (STQ.Empty = FALSE) **Eventually** (S_Schd);
After (S_Schd) **Never** (C_Intr+A_Intr+S_Schd+N_Schd+R_Intr) **UntilAfter** (S_Ret);

After (NTQ.Empty = FALSE) **Eventually** (N_Schd);
After (N_Schd) **Never** (C_Intr+A_Intr+S_Schd+N_Schd+R_Intr) **UntilAfter** (N_Ret);

After (RFM.Pending) **Eventually** (R_Intr);
After (R_Intr) **Never** (C_Intr+A_Intr+S_Schd+N_Schd+R_Intr) **UntilAfter** (R_Ret);

Figure 6.8: Properties of Sensor-to-Network component

component with the corresponding input message types of the Network (or Sensor) component as shown in Figure 6.7;

- Create an AIM system, SN, which consists of the following three stub AIM processes: (i) SP, whose input message types are *C_Intr*, *A_Intr*, *S_Schd*, *OP_Ack*, and *Done*, whose state model outputs messages of the types, *C_Ret*, *A_Ret*, *S_Ret*, *Data*, and *Sent_Ack*, and whose variables are *Pending* and *Empty*; (ii) NP, whose input message types are *N_Schd*, *R_Intr*, *Data*, and *Sent_Ack*, whose state model outputs messages of types *N_Ret*, *R_Ret*, *OP_Ack*, and *Done*, and whose variables are *Pending* and *Empty*; (iii) ENV, whose input message types are *C_Ret*, *A_Ret*, *S_Ret*, *N_Ret*, and *R_Ret*, and whose state model outputs messages of the types, *C_Intr, A_Intr, S_Schd, N_Schd*, and *R_Intr*;

- Execute the cone-of-influence analysis, the "enabled" function in Figure 6.1,

and the validity check of circular dependencies on the properties of the Sensor component and the Network component, and execute the cone-of-influence analysis on the assumptions in Figure 6.8, which leads to inclusion of the properties in Figure 6.9 into the abstraction.

Repeatedly (Data);
After(Data) **Never** (Data) **UntilAfter** (OP_Ack);

IfRepeatedly (Data) **Repeatedly** (RFM.Pending);
IfRepeatedly (Data) **Repeatedly** (Not RFM.Pending);
After (Data) **Eventually**(OP_Ack);
Never (OP_Ack) **UntilAfter** (Data);
After (OP_Ack) **Never** (OP_Ack) **UntilAfter** (Data);

Figure 6.9: Properties included in abstraction

We then checked the "repeated transmission" property on SN by following the steps given in Section 6.3.5. It is easy to observe that the property holds on the abstraction under the assumptions in Figure 6.8. Therefore, we concluded that the property also holds on the executable representation of the Sensor-to-Network component under the given assumptions.

6.4.4 Verification via Abstraction Refinement

An abstraction of a composed component may be refined by introducing, verifying, and enabling properties of the sub-components of the composed component or even by revising and re-verifying the sub-components. We illustrate how an abstraction is refined with the verification of *Property 6.4.1* on the Sensor-to-Network component. (Space limitations prohibit showing the formal specifications of the properties given hereafter.)

Property 6.4.1 *The Sensor-to-Network component transmits any hardware sensor reading exactly once.*

For checking *Property 6.4.1*, an abstraction of the Sensor-to-Network component was constructed. Model checking of *Property 6.4.1* on the abstraction returned false. By analyzing the error trace from COSPAN, we observed that the abstraction is too abstract to enable model checking of *Property 6.4.1* and has to be refined. To refine the abstraction, we introduced and checked *Property 6.4.2* on the Network component.

Property 6.4.2 *The Network component transmits any of its inputs exactly once assuming that a new input arrives only after it outputs a Sent message to indicate its last input has been successfully transmitted.*

Property 6.4.2 was successfully verified on the Network component, but it was not enabled in the composition of the Sensor-to-Network component. To enable the property on the Network component, we introduced and verified *Property 6.4.3* on the Sensor component.

Property 6.4.3 *The Sensor component outputs any hardware sensor reading exactly once and after an output, it will not output again until after a message of the type, Done, is received.*

The verification of *Property 6.4.3* on the sensor component returned false due to a bug of the Sensor component. In the Sensor component, each time a hardware sensor reading is put in the output buffer, a thin thread [30] is created to output the data. There is a flag that should be set when a sensor reading has been output and a *Done* message has not been received. However, the thin thread fails to set the flag

correctly. When the physical sensor outruns the physical network, since the output flag is not set, the sensor component may output again before it receives the *Done* message for its last output. This violates the second assertion in *Property 6.4.3*.

The bug was corrected and all properties of the Sensor component, including *Property 6.4.3*, were re-verified. A new Sensor-to-Network component was composed from the corrected Sensor component and the Network component. An abstraction of the newly composed component was constructed, on which *Property 6.4.1* was successfully verified.

Remarks: In this example, the verification of *Property 6.4.1* on the Sensor-to-Network component requires introducing and verifying additional properties of its sub-components, which is only for the purpose of demonstrating abstraction refinement and does not indicate that we frequently need to introduce and verify additional properties of a component. Since our approach targets software product lines, careful domain analysis and a few reuses often could lead to a quite stable set of properties for a component.

6.5 Analysis of Case Study

Application of our approach to integration of model checking into CBD to the TinyOS components leads to the detection of a coordination error which is related to component composition, and a significant reduction in model checking complexity.

6.5.1 Detection of Coordination Error

By model checking of the "repeated transmission" property and *Property 1* on the Sensor-to-Network component, we have detected a coordination error as described

in Section 6.4.4. This error would be hard to detect with conventional test-case based testing methods.

6.5.2 Model Checking Complexity Reduction

Direct verification of a property on a composed component with model checking is often infeasible due to state space explosions. In our approach, model checking of a property on a composed component is reduced to three sub-tasks: model checking of the properties of the sub-components, construction and refinement of an abstraction of the component, and model checking of the property on the abstraction. Complexities of these sub-tasks are often significantly lower than the complexity of directly model checking the property on the component. Verified properties of the sub-components can often be reused. The complexity of model checking a newly introduced property of a sub-component is lower since the sub-component has a smaller state space, and may be further reduced if the sub-component is also a composed component. Since the abstraction construction usually only involves a few environment assumptions and verified sub-component properties, it often has a modest complexity. Although the abstraction refinement may require user interactions, it is expected to be facilitated by domain-specific knowledge. An abstraction of a component only captures the aspect of the component required for verification of a specific property and usually consists of a few simple automata, therefore the verification of the property on the abstraction finishes fairly fast.

We illustrate the reduction attained in our approach on model checking complexity with the statistics from the TinyOS case study. Table 6.1 shows four model checking runs for verifying the "Repeated transmission" property on the Sensor-

Run	Component	Time	Memory
1	Sensor-to-Network	89m15.45s	208.48M
2	Sensor	10m41.01s	33.673M
3	Network	18.0s	6.8239M
4	Abstraction of SN	0.1s	0.1638M

Table 6.1: Verification complexity comparison

to-Network component. Run 1 checks the property on the composed component directly for comparison purposes. Run 2 (or Run 3, respectively) checks the properties in Figure 6.4 (or Figure 6.6) on the Sensor (or Network) component. Run 4 checks the "Repeatedly transmission" property on the abstraction of the Sensor-to-Network component. The complexities for model checking the sub-components and the abstraction are an order-of-magnitude lower than the complexity of directly checking the composed component. Furthermore, the verification results for the Sensor and Network components were reused from previous studies. The statistics shown in Table 6.1 only involves one level of composition. In a multi-level composition, this approach can model check higher level composed components that cannot be directly model checked due to state space explosions.

6.6 Related Work

There has been extensive research [53, 1, 3, 46, 4, 19] on compositional reasoning in the formal methods community. Most prior work applies compositional reasoning in a top-down approach: To check properties of a large system, the system is decomposed into modules recursively in a top-down fashion. Our research is based on the prior work, but combines the top-down approach with the bottom-up com-

ponent composition process of CBD. Properties of components are verified as they are composed from simpler components in a bottom-up fashion and verification of these properties is based on compositional reasoning.

A closely related work to our research is Compositional Reachability Analysis (CRA) by Graf and Steffen [27], Yeh and Young [71], Cheung and Kramer [11, 12, 13, 10, 14], et al. CRA analyzes a system or its modules in the context of the system. Modules are represented by Labeled Transition Systems (LTSs) or similar compositional representations of state space graphs. It is assumed that there exist LTSs of the lowest level modules. The LTS of a higher level module is composed from LTSs of its sub-modules and is minimized according to the property to be checked and context constraints. Property decomposition has not yet been supported in CRA, therefore a property involving multiple modules may lead to a complex global LTS. Another related work is Modular Feature Verification by Fisler and Krishnamurthi [21], which targets systems developed by Feature-Oriented Programming (FOP). In FOP, components are features that are orthogonal to the component concepts used in this chapter. Our approach supports both component verification in the context of a system, and component verification with only environment assumptions and without a specific composition context. A component is represented by a set of verified temporal properties. Properties of a primitive component are obtained by directly model checking its original representation that can be in any model-checkable languages such as model-checkable subsets of UML, SDL, JAVA, or C/C++. Properties of a composed component are obtained by model checking its abstractions constructed from its environment assumptions and verified properties of its sub-components. Our approach also supports property decomposition by

applying top-down compositional reasoning.

There is also related research on automatic generation of assumptions in compositional reasoning, such as [23]. [23] proposes an approach to automatic generation of assumptions for safety properties in the context of representing systems and their components using LTSs. In this approach, an automatically generated assumption can be quite complex and hard to check on other components.

6.7 Summary

An approach to integration of model checking into the CBD of software systems has been presented. The case study on TinyOS demonstrates the applicability of this approach, the detection of a coordination error, and a significant reduction in model checking complexity. The reduction in model checking complexity seems scalable. This approach can be readily applied on many software computation models.

We have further scaled model checking of component-based systems by exploring the synergy between MDD and CBD. Components are developed following a combination of MDD and CBD: specifying the executable representations of components using an executable design language, in our case, xUML. We have realized the bottom-up approach to component verification based on model checking of software designs through translation. Properties of primitive components are model-checked on the xUML models of these components using the ObjectCheck toolkit. Properties of composite components are model-checked on the abstractions of these components also using the ObjectCheck toolkit.

Chapter 7

Conclusions and Future Work

Practical and scalable model checking of software systems demands seamless integration of model checking and software development. Model checkers must become an integral part of software development processes similar to the role played by debuggers. This dissertation research has successfully integrated model checking into two emerging software development processes, MDD and CBD, and furthermore into a comprehensive software development process that combines MDD and CBD. This combined software development process yields a well-structured representation of a software system at a level of abstraction where scalable model checking is feasible, and also leads directly to production software. The translation-based approach, taken in this research, enables reuse of much of the model checking apparatus for different representations at the MDD level and facilitates structured, orderly application of state space reduction algorithms. The effectiveness of the integration and supporting tools and methods has been demonstrated by their successful application to model checking of interesting systems such as a NASA robot controller, an

online ticket sale system, and the TinyOS run-time system for networked sensors. Section 7.1 summarizes the individual research results which justify these conclusions. The results of this research have also established a foundation for continued development of concepts and methods for application of model checking as a practical approach to verification of software systems for safety, security and reliability properties. Section 7.2 enumerates some of the possible future research directions enabled or suggested by the results of this research.

7.1 Summary of Contributions

In this dissertation, we have developed a comprehensive approach to integration of model checking into two emerging software development processes, MDD and CBD, and their combination. This approach addresses the two major challenges for practical and scalable model checking of software systems: *applicability* and *intrinsic complexities of model checking*, under the following framework: (1) bridging applicability gaps through automatic translation of software representations to directly model-checkable formal representations, (2) seamless integration of state space reduction algorithms in the translation via static analysis, and (3) scaling model checking capability and achieving state space reduction by systematically exploring compositional structures of software systems.

We have integrated model checking into MDD by applying mature model checking techniques to industrial design-level software representations through *automatic translation* of these representations to the input formal representations of model checkers [68, 69, 70]. In the translation, we have applied many state space reduction algorithms to software representations under an *integrated model and prop-*

erty translation framework [70] in which the translation of a model depends on the property to be checked and the state space reduction algorithms to be applied. We have developed a *translation-based* approach [67] to compositional reasoning of software systems, which simplifies the proof, implementation, and application of compositional reasoning rules at the software system level by reusing the proof and implementation of existing compositional reasoning rules for directly model-checkable formal representations. We have developed an *integrated state space reduction framework* [65], which systematically conducts a *top-down* decomposition of a complex software system into directly model-checkable pieces by exploring domain-specific knowledge. We have designed, implemented, and applied a *bottom-up* approach [66] to model checking of component-based software systems, which composes verified systems from verified components and integrates model checking into CBD. We have further scaled model checking of component-based systems by exploring the synergy between MDD and CBD, i.e., specifying components in executable design languages, and realizing the bottom-up approach based on model checking of software designs through translation.

7.2 Future Research Directions

A grand challenge in software engineering research is to provide methods and tools that (1) facilitate development of safe, secure, and reliable software systems of *increasing complexity* and (2) can be seamlessly integrated in the *routine development efforts* of software engineers. Our future research aims at technology advances towards address of this challenge and will be focused on developing such methods and tools and on extending them to hardware/software co-design and co-verification.

We will pursue our future research through exploring synergistic integration of systematic testing, formal methods, static program analysis, and run-time analysis. In the near future, We would like to focus our research in the following directions.

7.2.1 Scalable Verification of Component-Based Systems

Our approach to component verification will be extended to support verification of large-scale software components conforming to industrial standards such as CORBA and DCOM. This requires appropriate formalization of the component models supported by these standards and adequate tool supports for discharging direct verification of modest-size components that are specified in their native representations such as Java and C++.

In a future product line of software systems, besides the conventional component paradigm, emerging component paradigms such as feature-oriented programming [55] may also be involved. How to effectively verify a product line that involves several different component paradigms is an open problem.

7.2.2 Software Security Assurance via Formal Verification

Our previous research is applicable to security assurance of software systems. A sample application is security assurance of web service contracts. Web service contracts are widely used in both the academia and the business world, for instance, in forming virtual organizations over computational grids and in defining business-to-business solutions. These contracts are amenable to application of our previous research results since they have the following characteristics: first, they specify concurrency-intensive interactions among web services; second, they are often formulated using

high-level executable representations, such as *Business Process Execution Language for Web Services* [34]; third, they are highly modularized. Security is a great concern about these contracts, for instance, whether these contracts respect security policies of the participants. We am planning to investigate how security policies can be represented in an integrated fashion with these contracts, how security properties such as information flow security can be formulated on these contracts, and how these properties can be established through formal verification.

7.2.3 Domain Knowledge Driven State Space Reduction

The complexities of software systems keep growing, which makes state space reduction a long-lasting theme in software model checking. On the other hand, these systems become increasingly customized to specific application domains. Our previous work on integrated state space reduction and on model checking of component-based systems has demonstrated the importance of domain-specific knowledge in state space reduction. We am planing to further explore how a verification system can adapt to different application domains and utilize the knowledge about these domains to effectively conduct state space reduction.

7.2.4 Hardware/Software Co-Verification

High safety, security, and reliability requirements of embedded systems such as TinyOS demand a tight integration of co-verification capability into the design and development of such embedded systems. Our previous research on the ObjectCheck toolkit has established a foundation for hardware/software co-verification. When combined with FormalCheck [38], an industrial model checking tool for hardware

designs from Cadence Design Systems, ObjectCheck supports hardware/software co-verification where both hardware designs in VHDL or Verilog and software designs in xUML are translated into S/R and model checked by COSPAN in an integrated fashion. A clearly defined research project is to design and implement highly reliable networked sensors with a co-verification capability that integrates ObjectCheck and FormalCheck.

Bibliography

[1] Martin Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3):506–534, May 1995.

[2] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[3] Rajeev Alur and Thomas Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, July 1999.

[4] Nina Amla, Ellen. A. Emerson, Kedar S. Namjoshi, and Richard Trefler. Assume-guarantee based compositional reasoning for synchronous timing diagrams. In *Proc. of 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 465–479, Genova, Italy, April 2001.

[5] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. of 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–3, Portland, Oregon, USA, January 2002.

[6] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1), 1988.

[7] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, Laurent Mounier, and Joseph Sifakis. IF: An intermediate representation for SDL and its applications. In *Prof. of 9th International SDL Forum*, pages 423–440, Montreal, Quebec, Canada, June 1999.

[8] Marius Bozga, Susanne Graf, and Laurent Mounier. Automated validation of distributed software using the IF environment. In *Proc. of LACPV'2001 Logical Aspects of Cryptographic Protocol Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, Paris, France, July 2001.

[9] Kennedy Carter. *http://www.kc.com.* kennedy Carter, 2004.

[10] Shing-Chi Cheung, Dimitra Giannakopoulou, and Jeff Kramer. Verification of liveness properties using compositional reachability analysis. In *Proc. of 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 227–243, Zurich, Switzerland, September 1997.

[11] Shing-Chi Cheung and Jeff Kramer. Enhancing compositional reachability analysis with context constraints. In *Proc. of 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 115–125, Los Angeles, California, USA, December 1993.

[12] Shing-Chi Cheung and Jeff Kramer. Compositional reachability analysis of finite-state distributed systems with user-specified constraints. In *Proc. of 3rd*

ACM SIGSOFT Symposium on Foundations of Software Engineering, pages 140–150, Washington, D.C., USA, October 1995.

[13] Shing-Chi Cheung and Jeff Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):334–377, October 1996.

[14] Shing-Chi Cheung and Jeff Kramer. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 8(1):49–78, January 1999.

[15] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. of Logic of Programs Workshop*, pages 52–71, Yorktown Heights, New York, USA, May 1981.

[16] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. Symmetry reductions in model checking. In *Proc. of 10th International Conference on Computer Aided Verification (CAV)*, number 1427 in Lecture Notes in Computer Science, pages 147–158, Vancouver, British Columbia, Canada, June 1998.

[17] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[18] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Bandera: a source-level interface for model checking Java programs. In *Proc. of 22nd*

International Conference on Software Engineering, pages 762–765, Limerick, Ireland, June 2000.

[19] Willem-Paul de Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Non-compositional Proof Methods*. Cambridge University Press, 2001.

[20] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of International Conference on Software Engineering*, pages 411–420, Los Angeles, California, USA, May 1999.

[21] Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In *Proc. of 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 152–163, Vienna, Austria, September 2001.

[22] Hubert Garavel and Joseph Sifakis. Compilation and verification of LOTOS specifications. In *Proc. of the IFIP WG6.1 Tenth International Symposium on Protocol Specification, Testing and Verification*, pages 379–394, Ottawa, Ontario, Canada, June 1990.

[23] Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Assumption generation for software component verification. In *Proc. of 17th IEEE International Conference on Automated Software Engineering (ASE)*, pages 3–12, Edinburgh, Scotland, UK, September 2002.

[24] Stefania Gnesi, Diego Latella, and Mieke Massink. Model checking uml statechart diagrams using jack. In *Prof. of 4th IEEE International Symposium on High-Assurance Systems Engineering (HASE)*, pages 46–55, Washington, D.C., USA, November 1999.

[25] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods. In *Proc. of 5th International Conference on Computer Aided Verification (CAV)*, volume 697 of *Lecture Notes in Computer Science*, pages 438–449, Elounda, Greece, June 1993.

[26] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with pvs. In *Proc. of 9th International Conference on Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997.

[27] Susanne Graf and Bernhard Steffen. Compositional minimization of finite state systems. In *Proc. of 2nd International Conference on Computer Aided Verification (CAV)*, volume 531 of *Lecture Notes in Computer Science*, pages 186–196, New Brunswick, New Jersey, USA, June 1990.

[28] R. H. Hardin, Z. Har'El, and Robert P. Kurshan. Cospan. In *Proc. of 8th International conference on Computer-Aided Verification (CAV)*, New Brunswick, New Jersey, USA, July 1996.

[29] Klaus Havelund and Jens U. Skakkebak. Applying model checking in java verification. In *Proc. of 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*, pages 216–231, Toulouse, France, September 1999.

[30] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Proc. of 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Cambridge, Massachusetts, USA, November 2000.

[31] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[32] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering (TSE)*, 23(5):279–295, 1997.

[33] Gerard J. Holzmann and Margaret H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Transactions on Software Engineering (TSE)*, 28(4), 2002.

[34] IBM. *http://www.ibm.com/developerworks/library/ws-bpel*. IBM, 2004.

[35] ITU. *ITU-T Recommendation Z.100 (03/93) - Specification and Description Language (SDL)*. ITU, 1993.

[36] C. Kapoor and Delbert Tesar. A reusable operational software architecture for advanced robotics (OSCAR). *The University of Texas at Austin, Report to U.S. Dept. of Energy, Grant No. DE-FG01 94EW37966 and NASA Grant No. NAG 9-809*, 1998.

[37] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[38] Robert P. Kurshan. *FormalCheck User's Manual*. Cadence Design Systems, 1998.

[39] Robert P. Kurshan, Vladimir Levin, Marius Minea, Doron Peled, and Husnu Yenigun. Static partial order reduction. In *Proc. of 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1384 of *Lecture Notes in Computer Science*, pages 345–357, Lisbon, Portugal, March 1998.

[40] Robert P. Kurshan, Vladimir Levin, and Husnu Yenigun. Compressing transitions for model checking. In *Proc. of 14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 569–581, Copenhagen, Denmark, July 2002.

[41] Robert P. Kurshan, Michael Merritt, Ariel Orda, and Sonia R. Sachs. Modelling asynchrony with a synchronous model. *Formal Methods in System Design*, 15(3):175–199, November 1999.

[42] Vladimir Levin and Husnu Yenigun. Sdlcheck: A model checking tool. In *Proc. of 13th International Conference on Computer Aided Verification (CAV)*, volume 1680 of *Lecture Notes in Computer Science*, pages 378–381, Paris, France, July 2001.

[43] Johan Lilius and Ivan Paltor. vUML: a tool for verifying UML models. In *Proc. of 14th IEEE International Conference on Automated Software Engineering (ASE)*, pages 255–258, Cocoa Beach, Florida, USA, October 1999.

[44] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[45] Ken L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[46] Ken L. McMillan. A methodology for hardware verification using compositional model checking. *Cadence Design Systems Technical Reports*, 1999.

[47] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison Wesley, 2002.

[48] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J. Holzmann. Implementing statecharts in Promela/Spin. In *Prof. of 2nd Workshop on Industrial-Strength Formal Specification Techniques (WIFT)*, pages 90–101, Boca Raton, Florida, USA, October 1998.

[49] Kedar Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In *Proc. of 12th International Conference on Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 435–449, Chicago, Illinois, USA, July 2000.

[50] Object Management Group (OMG). *Unified Modeling Language Specification, Version 1.3*. OMG, 1999.

[51] Object Management Group (OMG). *http://www.omg.org/mda*. OMG, 2004.

[52] Doron Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, 1996.

[53] Amir Pnueli. In transition from global to modular reasoning about programs. *Logics and Models of Concurrent Systems*, 1985.

[54] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proc. of 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166, Lisbon, Portugal, March 1998.

[55] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *Proc of 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443, Jyväskylä, Finland, June 1997.

[56] Project Technology. *http://www.projtech.com*. Project Technology, 2004.

[57] Jean-Pierre Quielle and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proc. of Symposium on Programming*, pages 337–351, Torino, Italy, April 1982.

[58] SES. *Objectbench User Manual*. SES, 1996.

[59] Natasha Sharygina and James C. Browne. Model checking software via abstraction of loop transitions. In *Proc. of 6th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 2621 of *Lecture Notes in Computer Science*, pages 325–340, Warsaw, Poland, April 2003.

[60] Natasha Sharygina, James C. Browne, Fei Xie, Robert P. Kurshan, and Vladimir Levin. Lessons learned from model checking a nasa robot controller. *Journal of Formal Methods in System Design (FMSD)*, 2004.

[61] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[62] Antti Valmari. A stubborn attack on state explosion. In *Proc. of 2nd International Conference on Computer Aided Verification (CAV)*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165, New Brunswick, New Jersey, USA, June 1990.

[63] W3C. *http://www.w3.org/TR/wsdl*. W3C, 2004.

[64] Wenli Wang, Zoltan Hidvegi, Andrew D. Bailey Jr., and Andrew B. Whinston. E-processes design and assurance using model checking. *IEEE Computer*, 33(10):48–53, 2000.

[65] Fei Xie and James C. Browne. Integrated state space reduction for model checking executable object-oriented software system designs. In *Proc. of 5th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 2306 of *Lecture Notes in Computer Science*, pages 331–335, Grenoble, France, April 2002.

[66] Fei Xie and James C. Browne. Verified systems by composition from verified components. In *Proc. of 4th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 277–286, Helsinki, Finland, September 2003.

[67] Fei Xie, James C. Browne, and Robert P. Kurshan. Translation-based compositional reasoning for software systems. In *Proc. of 12th International Formal Method Europe (FME) Symposium*, volume 2805 of *Lecture Notes in Computer Science*, pages 582–599, Pisa, Italy, September 2003.

[68] Fei Xie, Vladimir Levin, and James C. Browne. Model checking for an exe-

cutable subset of uml. In *Proc. of 16th International Conference on Automated Software Engineering (ASE)*, pages 333–336, Coronado Island, San Diego, California, USA, November 2001.

[69] Fei Xie, Vladimir Levin, and James C. Browne. Objectcheck: A model checking tool for executable object-oriented software system designs. In *Proc. of 5th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 2306 of *Lecture Notes in Computer Science*, pages 331–335, Grenoble, France, April 2002.

[70] Fei Xie, Vladimir Levin, Robert P. Kurshan, and James C. Browne. Translating software designs for model checking. In *Proc. of 7th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 324–338, Barcelona, Spain, March 2004.

[71] Wei-Jen Yeh and Michal Young. Compositional reachability analysis using process algebra. In *Proc. of Symposium on Testing, Analysis, and Verification*, pages 49–59, Victoria, British Columbia, Canada, October 1991.

Vita

Fei Xie was born to Shengguang Xie and Qingying Wang in Taiyuan, Shanxi, P. R. China, and has two elder brothers, Hang and Xiang. He grew up in Luoyang, Henan, P. R. China. He received a B.S. in Computer Science from Northwestern Polytechnic University, Xi'an, Shaanxi, P. R. China in 1995. He received a M.S. in Computer Science from Tsinghua University, Beijing, P. R. China in 1998. He came to the University of Texas at Austin to pursue a Ph.D. in August, 1998.

Fei has been happily married to his lovely wife, Huaiyu Liu, since 1998.

Permanent Address: 1620 West 6th St. Apt. P

Austin, TX 78703 USA

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.