Translating Software Designs for Model Checking [⋆]

Fei Xie[1], Vladimir Levin[2], Robert P. Kurshan[3], and James C. Browne[1]

[1] Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712
Email: {feixie, browne}@cs.utexas.edu Fax: +1 (512) 471-8885
[2] Microsoft, One Microsoft Way, Redmond, WA 98052
Email: vladlev@microsoft.com Fax: +1 (425) 936-7329
[3] Cadence, 35 Spring St., New Providence, NJ 07974
Email: rkurshan@cadence.com Fax: +1 (908) 898-1435

Abstract. This paper presents a systematic consideration of the major issues involved in translation of executable design level software specification languages to directly model-checkable formal languages. These issues are considered under the framework of *integrated model/property translation* and include: (1) translator architecture; (2) semantics translation from a software language to a formal language; (3) property specification and translation; (4) transformations for state space reduction; (5) translator validation and evolution. Solutions to these issues are defined, described, and illustrated in the context of translating xUML, an executable design level software specification language, to S/R, the input formal language of the COSPAN model checker.

1 Introduction and Overview

Model checking [1, 2] has major potential for improving reliability of software systems. 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 design level software 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 [3–8], 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

[⋆] This research was partially supported by NSF grant 010-3725.

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 paper 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 [8] from xUML [9], an executable design level specification language, to S/R [10], the input language of the COSPAN [10] model checker. (Another translator [11], which translates SDL [12] 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 via 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/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 in translators.
- **Translator validation and evolution.** Translators must be validated for correctness. They must be able to adapt to evolution of source software languages 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 paper is organized as follows. In Sections 2, 3, 4, 5, and 6, we elaborate on these issues and discuss their solutions in the context of the xUML-to-S/R translator. We summarize several case studies using the xUML-to-S/R translator in Section 7, discuss related work in Section 8, and conclude in Section 9.

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.

2.1 A general architecture for translators

A general architecture for translators is shown in Figure 1. A notable feature of this

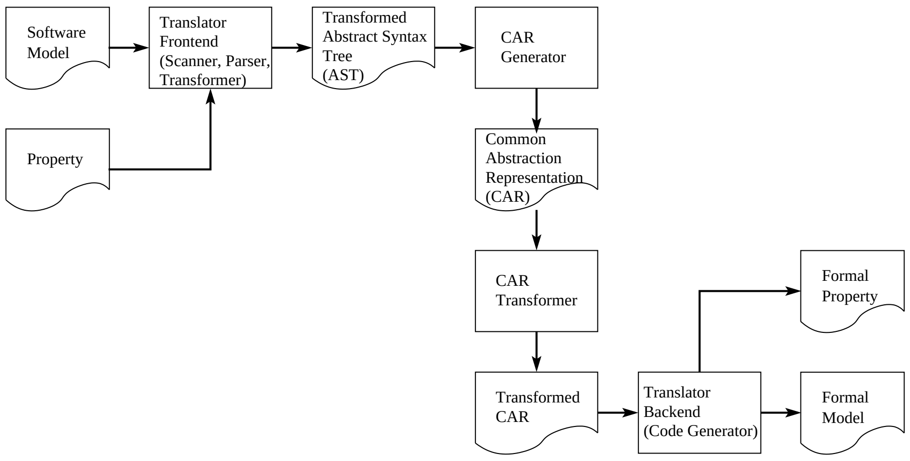
Software Model
Translator Frontend (Scanner, Parser, Transformer)
Transformed Abstract Syntax Tree (AST)
CAR Generator
Property
Common Abstraction Representation (CAR)
CAR Transformer
Formal Property
Transformed CAR
Translator Backend (Code Generator)
Formal Model

Fig. 1. Translator architecture

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 [13]. These transformations are partially guided by directives written in an annotation language to be discussed in Section 5.1. Functionalities of other components are discussed in Section 2.2 after we introduce the CAR.

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. Therefore, 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 [14] 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 Semantics Translation from Software Language to Formal Language

To translate a software language for model checking, a proper target formal language must be 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.

3.1 Selecting target formal language

There are many directly model-checkable formal languages. Promela [15], SMV [16], and S/R [10] are among the most widely used. These languages have various semantics. Their corresponding model checkers, SPIN [15], SMV [16], and COSPAN [10], 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*. These three factors were considered synergistically in selecting the target language for translating xUML.

– **Application domain.** While this paper 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, at least 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 of these algorithms 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. In addition, COSPAN supports a wide range of state space reduction algorithms such as localization reduction [10], static partial order reduction [14], and a prototype implementation [17] of predicate abstraction.

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

3.2 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 3.4) 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 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.

3.4 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 from the source language 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 asynchronous 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.

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.

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 [18] 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 [19].

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.

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.

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 [20] 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 support translation of a closed system that consists purely of entities specified in the software language, xUML in our case.

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.

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 software 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 [13].

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.

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 can be 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 algorithms can be found in [21].

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.

5.3 Static partial order reduction (SPOR)

Partial order reduction (POR) [22–24] 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 [22–24], 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 [14], 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.

5.4 Predicate abstraction

Predicate abstraction [25] 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 [17] 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.

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 [26]. 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 paper. 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.

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 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 [27], 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.

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.

7 Case Studies Using xUML-to-S/R Translator

The xUML-to-S/R translator has been applied in model checking designs of real-world software systems: a robot control system [28] from the robotics research group at the University of Texas at Austin, a prototype online ticket sale system [29], the TinyOS run-time environment [30] for networked sensors from University of California, Berkeley. The case study [31] on the robot control system demonstrated model checking of non-trivial software design models with the translator. In the case study [32] on online transaction systems, state space reduction capabilities of model transformations in the translator and interactions of these transformations were investigated. The TinyOS case study [33] demonstrated the translation support for compositional reasoning. Co-design and co-verification studies on TinyOS using the translator are in progress.

8 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 [34]. The vUML tool [7] 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 [35, 36] on verification of UML Statecharts by translating Statecharts into directly model-checkable languages. The CAESAR system [37] 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 [38] proposes IF [39], 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 paper extends the architecture for conventional compilers. Similar extensions have been proposed in [37, 38]. 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 [3] tool from Microsoft, the FEAVER [6] tool from Bell Labs, and the Bandera [4] 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.

9 Conclusions

Translation plays an increasingly important role in software model checking and enables reuse of mature model checking techniques. This paper identifies and formulates issues in translation for model checking of executable software designs. 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 support for model checking of other design level or implementation level software representations.

References

1. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. Logic of Programs Workshop (1981)
2. Quielle, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. 5th International Symposium on Programming (1982)
3. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. POPL (2002)
4. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Roby: Bandera: a source-level interface for model checking Java programs. ICSE (2000)
5. Havelund, K., Skakkebaek, J.: Applying model checking in Java verification. SPIN (1999)
6. Holzmann, G.J., Smith, M.H.: An automated verification method for distributed systems software based on model extraction. IEEE TSE **28** (2002)
7. Lilius, J., Porres, I.: vUML: a tool for verifying UML models. ASE (1999)
8. Xie, F., Levin, V., Browne, J.C.: ObjectCheck: a model checking tool for executable object-oriented software system designs. FASE (2002)
9. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model Driven Architecture. Addison Wesley (2002)
10. Hardin, R.H., Har'El, Z., Kurshan, R.P.: COSPAN. CAV (1996)
11. Levin, V., Yenigün, H.: SDLCheck: A model checking tool. CAV (2001)

12. ITU: ITU-T Recommendation Z.100 (03/93) - Specification and Description Language (SDL). ITU (1993)
13. Sharygina, N., Browne, J.C.: Model checking software via abstraction of loop transitions. FASE (2003)
14. Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigün, H.: Static partial order reduction. TACAS (1998)
15. Holzmann, G.J.: The model checker SPIN. TSE 23(5) (1997)
16. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
17. Namjoshi, K., Kurshan, R.P.: Syntactic program transformations for automatic abstraction. CAV (2000)
18. Kurshan, R.P.: FormalCheck User's Manual. (1998)
19. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. Formal Methods in Software Practice (1998)
20. de Rover, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Non-compositional Proof Methods. Cambridge Univ. Press (2001)
21. Kurshan, R.P., Levin, V., Yenigun, H.: Compressing transitions for model checking. CAV (2002)
22. Godefroid, P., Pirottin, D.: Refining dependencies improves partial-order verification methods. CAV (1993)
23. Peled, D.: Combining partial order reductions with on-the-fly model-checking. FMSD (1996)
24. Valmari, A.: A stubborn attack on state explosion. CAV (1990)
25. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. CAV (1997)
26. Xie, F., Browne, J.C., Kurshan, R.P.: Translation-based compositional reasoning for software systems. FME (2003)
27. Pnueli, A., Siegel, M., Singerman, E.: Translation valdation. TACAS (1998)
28. Kapoor, C., Tesar, D.: 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)
29. Wang, W., Hidvegi, Z., Bailey, A.D., Whinston, A.B.: E-processes design and assurance using model checking. IEEE Computer Vol. 33 (2000)
30. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. ASPLOS-IX (2000)
31. Sharygina, N., Browne, J.C., Kurshan, R.P.: Formal object-oriented analysis for software reliability design for verification. FASE (2001)
32. Xie, F., Browne, J.C.: Integrated state space reduction for model checking executable object-oriented software system designs. FASE (2002)
33. Xie, F., Browne, J.C.: Verified systems by composition from verified components. ESEC/FSE (2003)
34. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Computer Networks and ISDN Systems **14** (1988)
35. Gnesi, S., Latella, D., Massink, M.: Model checking UML statechart diagrams using JACK. HASE (1999)
36. Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.: Implementing statecharts in Promela/Spin. Industrial Strength Formal Specification Technologies (1993)
37. Garavel, H., Sifakis, J.: Compilation and verification of LOTOS specifications. PSTV (1990)
38. Bozga, M., Graf, S., Mounier, L.: Automated validation of distributed software using the IF environment. CAV (2001)
39. Bozga, M., Fernandez, J.C., Ghirvu, L., Graf, S., Krimm, J., Mounier, L., Sifakis, J.: IF: An intermediate representation for SDL and its applications. SDL-Forum'99 (1999)