Model Checking for an Executable Subset of UML

Fei Xie[1] Vladimir Levin[2] James C. Browne[1]

[1] Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712, USA
{feixie, browne}@cs.utexas.edu

[2] Computing Sciences Research Center
Bell Laboratories
Murray Hill, NJ 07974, USA
levin@research.bell-labs.com

Abstract

This paper presents an approach to model checking software system designs specified in xUML [3, 9], an executable subset of UML. This approach is enabled by the execution semantics of xUML and is based on automatic translation from xUML to S/R [2], the input language of the COSPAN [2] model checker. Model transformations are applied to reduce the state space of the resulting S/R model that is to be verified by COSPAN. An xUML level logic for specifying properties to be checked is defined. Automated support is provided for translating properties specified in the logic to S/R representations and mapping error traces generated by COSPAN to xUML representations.

1. Introduction

This paper presents an approach to model checking software system designs specified in xUML[3, 9], an executable subset of UML. The execution semantics of xUML include action semantics following a proposal for Action Semantics for UML [6] to OMG. The steps in the approach are:

a. A system design is specified in xUML as an executable model;
b. A property to be checked against the system design is specified in an xUML level logic;
c. The xUML model and the property are automatically translated to a model and a query in the S/R [2] automaton language by an xUML-to-S/R translator;
d. The S/R query is automatically checked against the S/R model by the COSPAN [2] model checker;
e. If a query fails, an error track is generated by COSPAN and is automatically translated into an error report in the name space of the xUML model.

The core of our approach is the comprehensive automation support for translation of an xUML model to an S/R model, for translation of a property specified in the xUML level logic to an S/R query, and for translation of an error track generated by COSPAN to an xUML representation. Model transformations leading to an S/R model with minimal state space form another major part of our work.

Research on model checking software systems has been mainly focused on system representations at either design level or programming language implementation level. Model checking designs facilitates early detection of design errors while model checking implementations [7, 11] may uncover errors introduced in the implementation phase. In this research, model checking is applied to executable designs which have not yet been implemented but for which implementations can be either automatically generated based on a predefined architecture or manually coded. Due to space limitations, we only compare with the most closely related work: the automatic verification tool for UML from the University of Michigan [1], and the vUML tool [5]. Both tools translate and verify UML models based on ad hoc execution semantics which did not include action semantics. Neither supports formulation of properties to be checked on the UML model level. We also addressed the translation of generalization relationships between classes in UML models, which is not addressed in [1, 5].

The rest of this paper is organized as follows: xUML, COSPAN, and S/R are sketched in Section 2. Section 3 presents major algorithms in the automatic translation from xUML to S/R. Model transformations reducing state spaces are discussed in Section 4. Section 5 introduces the automatic analysis support that facilitates model checking xUML models. Conclusions are given in Section 6.

2. Overview of xUML, COSPAN, and S/R

An xUML model consists of class definitions, inter-class relationships, and class instantiations. The execution behavior of a class instance is specified by a Moore state model

where each state has an associated action that is executed in a run-to-completion mode upon entry to the state. State transitions are invoked by messages. The execution behavior of an xUML model is an asynchronous interleaving of the executions of the state models of class instances.

COSPAN is a model checker with synchronous semantics and implements multiple state space reduction algorithms, one of which, Symbolic Verification, is not readily implementable in model checkers with asynchronous interleaving semantics. COSPAN also enables Partial Order Reduction [8] through Static Partial Order Reduction [4], integration of which with Symbolic Verification yields a potentially powerful state space reduction method.

In S/R, a system is composed of synchronously interacting processes. 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.

In the following, all "processes" refer to S/R processes.

3. Automatic xUML-to-S/R Translation

xUML models with asynchronous interleaving execution semantics, dynamic creation of class instances, and potentially unbounded state spaces are automatically translated to S/R models with synchronous parallel execution semantics, a static set of processes, and finite state spaces. Each class instance is translated to a process. The private message queue of each class instance is modeled by a process.

3.1. Simulating Asynchrony with Synchrony

A global scheduler, also modeled by a process, nondeterministically schedules a process from among all the processes modeling the class instances that are ready for an xUML state action or an xUML state transition. Only the scheduled process can perform an S/R state transition corresponding to a state action or a state transition in the state model of the corresponding class instance. All other processes modeling class instances follow a self-loop S/R state transition back to their current S/R states.

Asynchronous message passing between class instances is simulated by synchronous communication between processes modeling class instances and their message queues. Message types defined in a class are mapped to constants in the S/R model. These constants define an enumeration type which establishes the value range of the state variables that are declared in the processes modeling message queues of instances of the class and used to record the types of the messages kept in the queues.

3.2. Translating State Models

The behavior of a class instance is specified by its state model that consists of states, actions, and state transitions. Figure 1 illustrates a state from an xUML state model with its associated action and transitions.

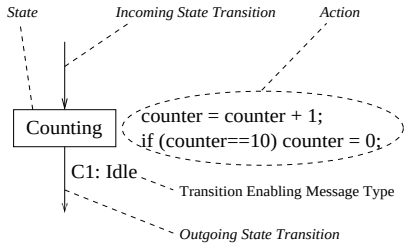

Figure 1. An Sample xUML State

To translate a state model, the translator first constructs the control flow graph of the state model. In the control flow graph, an action associated with a state is partitioned into primitive blocks. A primitive block consists of one or more sub-actions of the action. Two adjacent control points bracket either a primitive block or a state transition. Figure 2 illustrates the control flow graph segment corresponding to the state with its associated action and transitions in Figure 1. The primitive block between Control Point 1 and 2 in Figure 2, *counter = counter + 1*, consists of three sub-actions: a read, a plus, and a write.

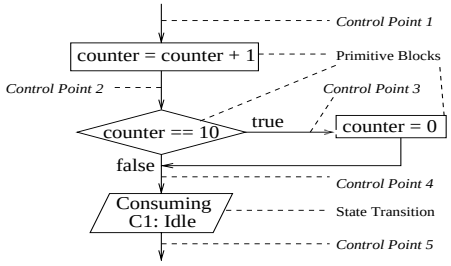

Figure 2. Control Flow Graph Segment

Partitioning of the action associated with a state into primitive blocks must preserve the run-to-completion se-

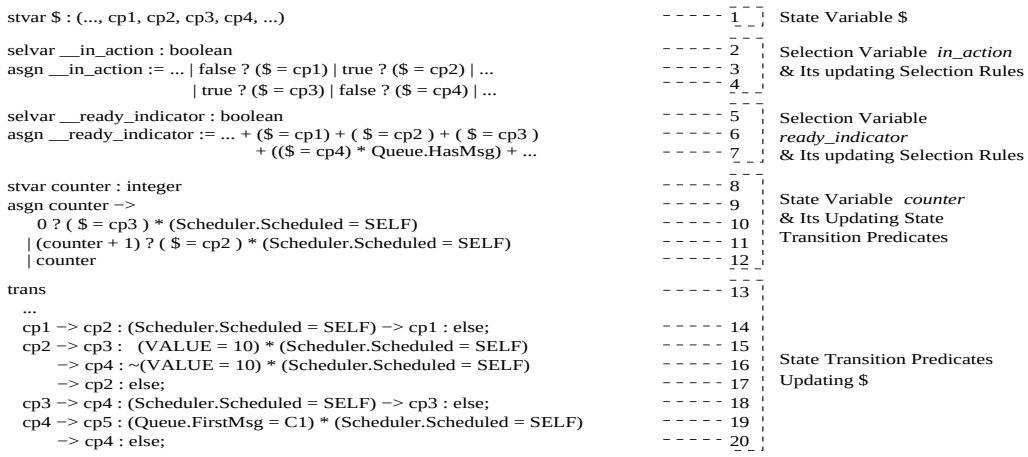

Figure 3. S/R Translation of the xUML State in Figure 1

mantics. For instance, all the primitive blocks between Control Points 1 and 4 in Figure 2 compose the action in Figure 1 and form a run-to-completion unit that must be executed without interruption. Therefore supplemental information is attached to the control points.

The state model, based on its control flow graph, is translated to semantic constructs of the process modeling the class instance as follows:

- A state variable $ of enumeration type is defined in the process and each control point in the control flow graph is one-to-one mapped to a value of $.
- The primitive block or state transition immediately following a control point is mapped to a set of state transition predicates or selection rules that depend on the value of $ corresponding to the control point.

The S/R process segments resulting from the state with its associated action and outgoing transition in Figure 1 are shown in Figure 3. For example, the primitive block, *counter = 0*, following Control Point 3 is mapped to two state transition predicates that are enabled when $ has the value of *cp3* and the process is scheduled by the global scheduler: One transition predicate sets the state variable corresponding to *counter* to 0 (Line 10); The other updates $ from *cp3* to *cp4* (Line 18). The outgoing state transition is mapped to a state transition predicate that updates $ from *cp4* to *cp5* when enabled (Line 19).

A process may take several selection/resolution cycles to perform a state action of the class instance it models if the action is partitioned into several primitive blocks. In order to guarantee the run-to-completion semantics of actions, a selection variable, *in_action*, is defined in the process, as shown in Figure 3. Once *in_action* is true, the process is scheduled continuously by the global scheduler until the process sets *in_action* to false. *in_action* is set to false if and only if $ has a value corresponding to a control point followed by a state transition or the first primitive block of an action; otherwise it is set to true.

3.3. Translating Generalizations

Under a generalization, subclasses may inherit attributes and message types from superclasses. The superclass attributes inherited by subclasses are mapped to state variables of the processes modeling the subclass instances. The superclass message types are mapped to constants which are included in the value ranges of the state variables that record the message types in the processes modeling the message queues of the subclass instances. Therefore no change is required on the translation of either a messaging action that sends a message of a superclass message type to an instance of a subclass or a state transition that consumes the message.

3.4. Handling Infinite or Dynamic State Spaces

The translator is able to translate xUML models with infinite or dynamic state spaces if designers can range infinite data types, numbers of class instances to be dynamically created, and infinite message queues by annotating the xUML models with an annotation language provided.

4. Transformations Reducing State Spaces

A significant part of our research is devoted to transforming xUML models before the translation to S/R in order to obtain S/R models with minimal state spaces.

4.1. Static Partial Order Reduction

The asynchronous interleaving semantics of xUML suggests applying Static Partial Order Reduction (SPOR) [4] to an xUML model prior to its translation into S/R. SPOR transforms the model by restricting its transition structure with respect to a property to be checked. Integrated application of Partial Order Reduction and Symbolic Verification is achieved by applying Symbolic Verification to the S/R model translated from the transformed xUML model.

4.2. Identification of Static Attributes

Class instances may have static attributes whose values never change during system execution. We implemented a labeling algorithm that tags static attributes when the xUML model is parsed. Instead of being translated to state variables, static attributes are translated into constants which do not contribute to the state space.

4.3. Identification of Self Messages

A self message is a message that a class instance sends to itself. The messaging action sending a self message and the transition consuming the message are translated as a whole to a single state transition predicate if the execution order of actions and state transitions can be preserved. An S/R state transition resulting from the state transition predicate has the same effect as sending and consuming the message.

4.4. Support for Symbolic Verification

To symbolically verify an S/R model, an explicit value range must be provided for each variable in the S/R model. These ranges can be provided by designers using the annotation language mentioned in Section 3.4. We are exploring transformations leading to S/R models whose state spaces can be reduced more effectively by Symbolic Verification.

5. Analysis Support and Tools

5.1. xUML Level Property Specification

We defined an xUML level property specification logic, provided an interface for specifying xUML level properties in the logic, and implemented a translator from xUML level properties to S/R queries. A property formulated in the logic consists of declarations of propositional logic predicates over xUML model constructs and declarations of temporal predicates. The temporal predicates are declared by instantiating a set of templates. A template consists of a temporal logic operator and a pattern of arguments. Each temporal predicate is an instantiation of a template where each argument is a propositional logic expression composed from the previously declared propositional predicates.

5.2. Post-Processing of Error Tracks

When a query on an S/R model fails, COSPAN generates an error track specifying an execution trace that is inconsistent with the query. A translator that automatically maps the error track to an error report in the xUML notation is provided. The error report consists of an execution trace of the corresponding xUML model, which violates the corresponding xUML level property.

6. Conclusions

We present an approach to model checking xUML models of complex and concurrent software systems. The approach is enabled by the executable nature of xUML, based on automatic translation of asynchronous xUML models to synchronous S/R models, and enhanced by xUML model transformations leading to S/R models with manageable state spaces. A long version of this paper with full details of the translation algorithms can be found in [12]. A successful application of this approach has been reported in [10].

7. Acknowledgments

We would like to acknowledge Robert P. Kurshan for his important role in initiating, supporting, and collaborating on this project. We would also thank Natasha Sharygina and Husnu Yenigün for their support. This research was partially supported by TARP grant 003658-0508-1999.

References

[1] K. Compton, Y. Gurevich, J. K. Huggins, and W. Shen. An Automatic Verification Tool for Uml. *Univ. of Michigan, EECS Dept. Tech. Report CSE-TR-423-00*, 2000.

[2] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. *Proc. of 8th International Conference on Computer Aided Verification*, 1996.

[3] Kennedy Carter. *http://www.kc.com/html/xuml.html*.

[4] R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. Static Partial Order Reduction. *Proc. of 4th International Conference Tools and Algorithms for the Construction and Analysis of Systems*, 1998.

[5] J. Lilius and I. Porres. vUML: a Tool for Verifying UML Models. *Proc. of the Automatic Software Engineering Conference*, 1999.

[6] OMG. *Action Semantics for the UML*. OMG, 2000.

[7] D. Y. W. Park, U. Stern, J. U. Sakkebak, and D. L. Dill. Java Model Checking. *Proc. of the Automatic Software Engineering Conference*, 2000.

[8] D. Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. *Formal Methods in System Design*, (8), 1996.

[9] Project Tech. *http://www.projtech.com/pubs/xuml.html*.

[10] N. Sharygina, R. P. Kurshan, and J. C. Browne. A Formal Object-oriented Analysis for Software Reliability. *Proc. of Fundamental Approaches to Software Engineering*, 2001.

[11] W. Visser, K. Havelund, G. Brat, and S. J. Park. Model Checking Programs. *Proc. of the Automatic Software Engineering Conference*, 2000.

[12] F. Xie, V. Levin, and J. C. Browne. Model Checking for an Executable Subset of UML. *Univ. of Texas at Austin, UTCS Tech. Report TR-01-34*, 2000.