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 an executable subset of UML, xUML. The approach is enabled by the execution semantics of xUML and is based on automatic translation from xUML to S/R [5], the input language of the COSPAN [5] model checker. Translation algorithms are defined and described, which cover class models, state models of classes, actions associated with states in state models, execution semantics, etc. The translation attempts 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. The approach is illustrated by some results from a verification study of a simplified robot control system.

1 Introduction and Overview

The Unified Modeling Language (UML) [14] is the standard representation of Object-oriented software system designs. However, the actions associated with states or transitions in standard UML statecharts are specified only as uninterpreted strings. Therefore, the UML model of a system design is neither executable nor compilable to an executable program in a programming language. As a result validation and verification of the system design has to be deferred until the system (or at least its components) is implemented in a programming language.

The Object Management Group (OMG) has finalized a proposal for Action Semantics for UML [15], submitted by the Action Semantics Consortium [1]. Combining the proposed action semantics and an appropriate subset of UML defines a specification language for executable Object-oriented software system designs. There are commercial products that implement subsets of UML with action semantics following the proposed action semantics. xUML [6, 18] is one of these subsets. An xUML model representing a software system design is executable; therefore it can be tested for both functionality and performance via simulation, and also can be formally verified through model checking. This paper presents an approach to model checking xUML models, gives the details of the automatic translation of xUML models to S/R [5] models that can be verified by the COSPAN [5] model checker, and illustrates the approach with some results from a verification study of a simplified robot control system. The steps in the approach are:

a. A system design is specified in xUML as an executable model;
b. The xUML model is tested by execution with a discrete event simulator;
c. A property to be checked against the system design is specified in an xUML level logic;
d. The xUML model and the property are automatically translated to a model and a query in the S/R automaton language by an xUML-to-S/R translator;
e. The S/R query is automatically checked against the S/R model by COSPAN;
f. 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.

Steps c, d, and f are the subject of this paper.

The model checker, COSPAN, implements the automata-theoretic approach to model checking [7]. Under this approach, a system is modeled by an L-ω-automaton [7] P, where P is represented as a synchronous parallel composition $P = P_1 \otimes P_2 \otimes \ldots \otimes P_k$ of components (all modeled as L-ω-automata). The query to be checked is also represented by an L-ω-automaton T. Verification consists of the automata language containment test $\mathcal{L}(P) \subset \mathcal{L}(T)$, whether the language of P is contained in the language of T.

We have selected COSPAN, which has synchronous parallel semantics, as our verification engine because

COSPAN implements multiple state space reduction algorithms and one of these algorithms, Symbolic Verification [12], is not readily implementable in model checkers with asynchronous interleaving semantics. COSPAN also supports application of Static Partial Order Reduction [8], which enables integrated application of Partial Order Reduction [20, 4, 17] that is readily applicable to asynchronous automata and Symbolic Verification that is readily applicable to synchronous automata (See Sect. 4.1). We will, in a later paper, analyze the circumstances where each state space reduction algorithm is most effective. COSPAN, because of its parallel execution semantics, may also offer advantages for model checking system designs with true parallel semantics.

The core of our approach is the automatic translation of xUML models with asynchronous interleaving execution semantics, dynamic creation and deletion of class instances, and potentially unbounded state spaces to S/R models with synchronous parallel execution semantics, a static set of interacting processes, and finite state spaces. Since the sizes of the state spaces of the S/R models directly affect the efficiency of model checking, model transformations towards S/R models with minimal state spaces 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 [16, 21] 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 manually coded or automatically generated by direct compilation based on a predefined architecture. Due to space limitation, we only compare with the most closely related research. We judge the most closely related research to be the automatic verification tool for UML from the University of Michigan [2], and the vUML tool [11]. 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 translation of generalization relationships between classes in UML models. There is also previous work on verifying UML statecharts [3, 9, 13] by translating statecharts into verifiable languages.

The rest of this paper is organized as follows: in Section 2 a brief overview of the semantics of xUML and S/R is given. Section 3 presents the major algorithms in the automatic translation from xUML to S/R. Model transformations towards state space reduction are discussed in Section 4. Section 5 introduces the automatic analysis support that facilitates model checking xUML models. Section 6 illustrates our approach by model checking the xUML model of a simplified robot control system. Conclusions and future work are given in Section 7.

2 Semantics of xUML and S/R

The semantic gap between asynchronous xUML models and synchronous S/R models makes the xUML-to-S/R translation a significant translation process. To facilitate easier understanding of the translation algorithms, we briefly sketch the semantics of xUML and S/R.

2.1 xUML Semantics

Currently our translator is able to translate a significant subset of xUML. Modeled under the current translatable subset, 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.

2.1.1 State Models

Behaviors of the instances of an active class are specified by a state model that consists of:

- States. A state is a stage in the state model.
- Message types. Each message type defines a kind of messages that instances of the class can receive during system execution.
- Transitions. A transition specifies which new state is achieved when an instance of the class in a given state receives a message of a particular type.
- Actions. An action is an activity or operation that is associated with a state and must be executed when an instance arrives in that state.

2.1.2 Actions

Actions can be divided into five categories as follows:

- Read or write actions that read or write attributes of class instances, or dynamically create or delete class instances.
- Computation actions that perform various mathematical calculations.
- Messaging actions that send messages to active class instances.
- Composite actions that are control structures and recursive structures that permit complex actions to be composed from simpler actions.
- Collection actions that apply other actions to collections of elements, avoiding explicit indexing and extracting of elements from these collections.

2.1.3 Generalizations

Under a generalization, subclasses inherit the 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.

2.1.4 Execution Semantics

The execution semantics of xUML is an asynchronous interleaving semantics with the following properties:

- Creation and Deletion of Class Instances. Class instances can be created either statically at system initialization, or dynamically during system execution. An active class instance having a born-to-die state model deletes itself when it enters a termination state. A passive class instance can be deleted by actions executed by active class instances.
- Asynchronous Message Passing. Active class instances communicate with each other through asynchronously message passing. Every active class instance has a private message queue that is FIFO and infinite. All messages directed to an active class instance are kept in its message queue after being generated and before being consumed.
- Active Class Instance Scheduling. An active class instance is ready to be scheduled for execution if either it has entered its current state and is ready to execute the associated action of the current state, or it has finished the action of its current state and is ready to perform a state transition that is enabled by the first message in its private message queue.

  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. Both the execution of an action and the performance of a state transition are 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. The message disposition rules are recorded in the message disposition table of each state model.

2.2 S/R Semantics

In the S/R automaton language, a system is composed of synchronously interacting processes. In the following discussion, all "processes" refer to S/R processes. A process represents an L-ω-automaton and consists of:

- State Variables. The current state of the process is determined by the current values of all its state variables. The state space of the process is bounded by the ranges of all its state variables.
- Selection Variables. Selection variables define selections, the outputs of the process. At each state, the value of a selection variable is nondeterministically selected from a set of values possible in that state.
- Inputs. Each process imports a subset of all the selection variables of other processes as its inputs.
- State Transition Predicates. State transition predicates specify how the process changes its state by updating its state variables as functions of its current state, selection variables, and inputs.
- Selection Rules. Selection rules assign values to selection variables as functions of state variables. Such a function is nondeterministic if several values are possible for one selection variable in some state.

2.2.1 Selection/Resolution Model

The system execution model of S/R, namely the "selection/resolution" model [7], is a clock-driven synchronous execution model, under which a system of processes behaves in a two-phase procedure every logical clock cycle, as shown in Figure 1:

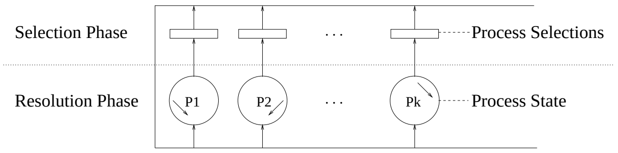

Figure 1: Selection/Resolution Model

- [1: Selection Phase] Each process "selects" a value possible in its current state for each of its selection variables. The values of all the selection variables of all the processes form the global selection of the system.
- [2: Resolution Phase] Each process "resolves" the current global selection by updating its state variables upon enabled state transition predicates and moving to a new state.

The communication between processes is synchronous: every process posts its selections through its selection variables in the selection phase of a clock cycle and inputs the selections from other processes in the resolution phase of the same clock cycle.

3 Automatic Translation of xUML Models to S/R Models

The automatic translation of an xUML model must yield an S/R model that is not only semantically faithful to the xUML model but also with a finite and fixed state space. The translation algorithms are given in detail since the translation is different from previous translations in that it targets the S/R automaton language with a synchronous execution model and includes the translation of action semantics. The algorithms enable model checking of substantial and significant xUML models.

3.1 Translating Class Instances to Processes

A class instance, either active or passive, is translated to a process. The private message queue of every active class instance is modeled by a separate process. Attributes of a class instance are translated to the state variables of the process corresponding to the class instance.

3.2 Simulating Asynchrony with Synchrony

To translate xUML models to S/R models, we simulate asynchronous execution semantics of xUML models with synchronous execution semantics of S/R models.

3.2.1 Modeling Asynchronous Message Passing

Asynchronous message passing between active class instances is simulated by synchronous communication between processes through modeling the private message queue of every active class instance as a separate process. Let processes IP_1 and IP_2 model two active class instances and processes QP_1 and QP_2 model their corresponding private message queues. A message, m, is sent from IP_1 to IP_2 asynchronously as shown in Figure 2:

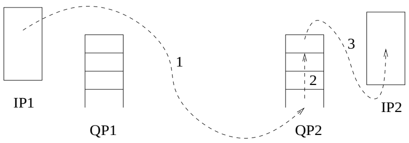

Figure 2: Modeling Asynchronous Communication

- [1: $IP_1 \to QP_2$] 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: $QP_2 \to IP_2$] QP_2 passes m to IP_2 through synchronous communication.

Message types defined in an active 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.

3.2.2 Modeling Asynchronous Execution

The asynchronous execution semantics of xUML is simulated by the synchronous execution semantics of S/R:

- Every process modeling an active class instance has a selection variable, *ready*, which indicates whether the active object instance modeled is ready for executing an xUML action, or performing an xUML state transition.
- A global scheduler, also modeled by a process, inputs the *ready* variables from all the processes modeling active class instances. When a rescheduling occurs, the global scheduler nondeterministically schedules a process from among all the processes modeling the active class instances that are ready. The global scheduler has a selection variable, *scheduled*, and the current value of *scheduled* indicates which active class instance is currently scheduled.
- All the processes modeling active class instances input *scheduled* from the global scheduler. Only the process that models the scheduled active class instance can perform an S/R state transition corresponding to an xUML action or an xUML state transition in the state model of the scheduled active class instance. All other processes modeling active class instances follow a self-loop S/R state transition back to their current S/R states.

3.2.3 Handling Unexpected Messages

When a process modeling the message queue of a class instance encounters a message that is to be ignored, it dequeues and discards the message. During model checking, a system error flag caused by an unexpected message will be caught by an automatically generated safety property. The translator generates such a safety property based on the message disposition tables for state models.

3.3 Translating State Models

The behavior of an active class instance is specified by its state model that consists of states, actions, and state transitions. Figure 3 illustrates a state from an xUML state model with its associated action and transitions.

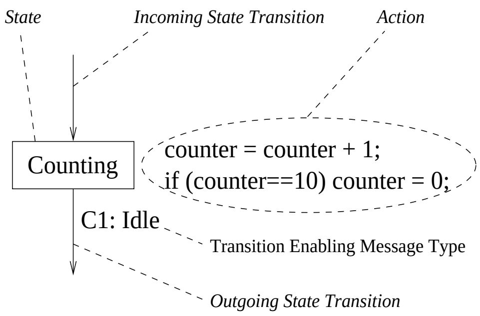

Figure 3: An Sample xUML State

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 4 illustrates the control flow

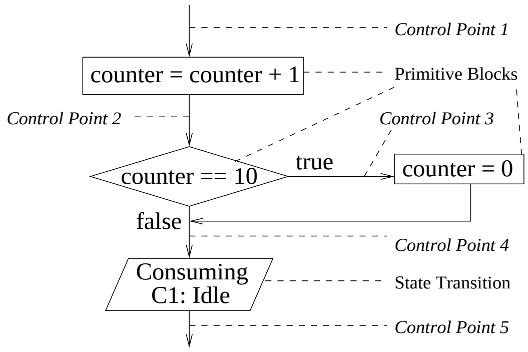

Figure 4: Control Flow Graph Segment

graph segment corresponding to the state with its associated action and transitions in Figure 3. The primitive block between Control Point 1 and 2 in Figure 4, *counter = counter + 1*, consists of three sub-actions: a read action, a plus action, and a write action.

Partitioning of the action associated with a state into primitive blocks must preserve the run-to-completion semantics. For instance, all the primitive blocks between Control Points 1 and 4 in Figure 4 compose the action in Figure 3 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 active 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 in the value range 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 3 are shown in Figure 5. 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 9); The other updates $\$$ from *cp3* to *cp4* (Line 16). The outgoing state transition is mapped to a state transition predicate that updates $\$$ from *cp4* to *cp5* when enabled (Line 17).

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 5. 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 following by a state transition or the first primitive block of an action; otherwise it is set to true.

3.4 Translating Actions

Computation actions of xUML are straightforwardly translated to their S/R counterparts. The translation of actions of other types is elaborated below.

3.4.1 Read or Write Actions

Intra-instance attribute reads (writes) are mapped to references to (state transition rules for) the corresponding state variables.

An inter-instance attribute read are simulated as follows: The process modeling the owner of the attribute outputs the value of the attribute through one of its selection variables and the selection variable is input by the process modeling the reader.

Translation of inter-instance attribute writes is more complex because state variables of a process cannot be directly updated by other processes. Let the process segment in Figure 5 belong to a process, PX, which models a class instance, X. Let the attribute of X, modeled by *counter*, be accessed by a write action from another class instance, Y, and the control point before the write action is *cp*. The inter-instance write action is simulated as follows: A selection variable, *counter_Y*, is defined in the process, PY, which models Y. A state transition rule updating *counter*, *asgn counter -> PY.counter_Y ? (Scheduler.Selection = PY) * (PY.\$ = cp)*, is added to PX. When $PY.\$=cp$ and PY is scheduled, in the next clock cycle PY sets *counter_Y* to the value to be written to *counter*. In the same clock cycle, PX instead of doing a self-loop state transition, resolves the state transition rule and sets *counter* to the value of *counter_Y* that is input by PX from PY. This is an extension to the scheduling rules in Section 3.3.2, which enables the translation of inter-instance writes.

3.4.2 Messaging Actions

A messaging action is mapped to a state transition predicate and a set of selection rules. The state transition predicate updates $\$$ from the control point immediately before the action to the control point immediately after

```
stvar $ : (..., cp1, cp2, cp3, cp4, ...)

selvar __in_action : boolean
asgn __in_action := ... | true ? ($ = cp1) | false ? ($ = cp4) | ...

selvar __ready_indicator : boolean
asgn __ready_indicator := ... + ($ = cp1) + ( $ = cp2 ) + ( $ = cp3 )
     + (($ = cp4) * Queue.HasMsg * Scheduler.None_in_action) + ...

stvar counter : integer
asgn counter ->
    0 ? ( $ = cp3 ) * (Scheduler.Scheduled = SELF)
   | (counter + 1) ? ( $ = cp2 ) * (Scheduler.Scheduled = SELF)

trans
  ...
  cp1 -> cp2 : (Scheduler.Scheduled = SELF) -> cp1 : else;
  cp2 -> cp3 :   (VALUE = 10) * (Scheduler.Scheduled = SELF)
       -> cp4 : ~(VALUE = 10) * (Scheduler.Scheduled = SELF)
       -> cp2 : else;
  cp3 -> cp4 : (Scheduler.Scheduled = SELF) -> cp3 : else;
  cp4 -> cp5 : (Queue.FirstMsg = C1) * (Scheduler.Scheduled = SELF)
       -> cp4 : else;
  ...
```

State Variable $

Selection Variable *in_action*
& Its updating Selection Rules

Selection Variable
ready_indicator
& Its updating Selection Rules

State Variable *counter*
& Its Updating State
Transition Predicates

State Transition Predicates
Updating $

Figure 5: S/R Translation of the xUML State in Figure 3

the action. The selection rules output the message and a synchronization signal through selection variables. The synchronization signal enables the process modeling the message queue of the receiver to get the message in the same clock cycle when the selection rules are enabled.

3.4.3 Composite Actions

There are three kinds of composite actions: group actions, conditional actions, and loop actions. A group action is composed of a sequence of sub-actions and is partitioned into one or more primitive blocks that are translated respectively. The translator also generates state transition predicates that advance *$* from one primitive block to the next primitive block.

A conditional action is composed of a test, which is mainly a computation action, and several branches. Every branch is a group action and translated as discussed above. The test is translated to a set of state transition predicates that lead to the S/R translations of these branches according to the result of the test.

The loop action provides repeated execution of a contained action so long as a test results in an appropriate value. The test is translated into two state transition predicates: Depending on the test result, one leads to the S/R translation of the contained action and the other exits the loop action.

3.4.4 Collection Actions

A collection action can be sequential or parallel. A sequential collection action applies a sub-action on elements of a collection in sequence. It is unfolded into a loop action with a test checking whether there are still untouched elements in the collection, and the sub-action as the contained action. The resulting loop action is translated as discussed above. A parallel collection action applies a sub-action on elements of a collection in parallel. Translation of the sub-action is extended so that all elements are processed simultaneously.

3.5 Translating Generalizations

Under a generalization, subclasses may inherit attributes and message types from superclasses. The superclass attributes inherited by subclasses are also mapped to state variables of the processes modeling the subclass instances.

The superclass message types are also 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. This solution requires no change to 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 such a message.

3.6 Guaranteeing Finite and Fixed State Spaces

Most model checkers including COSPAN, require that the models to be checked have finite and fixed state spaces. Our translator can translate xUML models with infinite or dynamic state spaces if necessary information is provided by designers through annotating the xUML models with an annotation language provided.

3.6.1 Ranging Data Types

A continuous infinite data type, like the float type, is discretized and represented by an integer interval type. COSPAN assumes every integer variable without an explicitly given value range lies in a default range.

3.6.2 Simulating Class Instance Dynamics

If instances of a class C, can be dynamically created and deleted during system execution, the dynamic creation and deletion is simulated as follows:

- An upper bound, N, on the number of instances of C that can co-exist at the same time during system execution, is estimated by system designers.
- The translator generates N processes, $P[0] \ldots P[N-1]$. Each $P[i]$, $0 \leq i < N$, models an instance of C.
- In each $P[i]$, $0 \leq i < N$, an additional state variable, *alive*, is used to indicate whether $P[i]$ is currently representing an existing instance of C. The *alive* variable of each P[i], $0 \leq i < N$, is initialized false.
- When an instance of C need to be created, some $P[j]$, $0 \leq j < N$, whose *alive* is false, is selected and *alive* of $P[j]$ is set to be true. $P[j]$ then participates in system execution by interacting with other processes.
- When an instance of C need to be deleted, *alive* of the corresponding $P[k]$, $0 \leq k < N$, is set to be false and $P[k]$ stops interacting with other processes.

3.6.3 Managing Message Queue Overflow

A message queue modeled by a process must have bounded size. This opens possibility of message queue overflow that may affect verification results. We deal with message queue overflow as follows:

- For each message queue, an upper bound on the number of messages that can be in the queue simultaneously is set by default or by system designers.
- Processes modeling the message queues are constructed based on the upper bounds.
- When the verification of a query reports false, an error track processing tool is used to analyze for actions that were trying to place a message into a full queue. When there is such a case, the verification will be invalidated and will be redone with a larger size for the message queue that was full.

4 Model Transformations towards State Space Reduction

State space reduction is critical to scalable application of model checking to xUML models. The state space complexities of the resulting S/R models directly affect the sizes of the xUML models that can be model checked. Therefore a significant part of our research is devoted to transforming xUML models before translation to S/R in order to get S/R models with minimal state spaces.

4.1 Static Partial Order Reduction

Different state space reduction algorithms are applicable in each of the several approaches to model checking. Symbolic Verification [12] is readily applied to synchronous automata while partial order reduction (POR) [20, 4, 17] is readily applied to asynchronous interleaving automata.

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 [20, 4, 17], 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 verification complexity.

The asynchronous interleaving semantics of xUML suggests application of Static Partial Order Reduction (SPOR) [8] 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 verified (For different properties, SPOR may translate an xUML model into different S/R models). This enables integrated application of POR while applying Symbolic Verification to the resulting S/R model.

4.2 Identification of Static Attributes

In xUML models, class instances may have static attributes whose values never change during system execution. We implemented a labeling algorithm that tags static attributes during the xUML model parsing phase. Instead of being translated to state variables, static attributes are translated into constants or selection variables 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 identified and 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 Transformations Supporting Symbolic Verification

In order to symbolically verify an S/R model with COSPAN, an explicit value range must be provided for every variable in the S/R model. An annotation language allowing designers to provide these ranges in xUML models has been implemented. We are also exploring transformations that lead to S/R models whose state spaces can be reduced more easily by symbolic verification.

5 Analysis Support and Tools

5.1 xUML-level Property Specification

Specification of properties to be checked is a critical factor in effective model checking. Effective model checking of xUML models by software engineers requires that the properties to be checked be specified in an xUML level logic. But COSPAN only accepts input queries formulated in S/R. Therefore we have defined an xUML level property specification logic, provided an interface for specifying xUML level properties in this logic, and implemented a translator for xUML level properties to S/R queries.

A property formulated in this 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 some template where each argument is a propositional logic expression built up from the previously declared propositional predicates. Space does not permit display of the full set of temporal logic templates but an example property is given in Section 6 and the full specification of the logic can be found on at the url, http://www.cs.utexas.edu/users/feixie/xUML.

5.2 Post-Processing of Error Tracks

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

6 Case Study

A real-world application used to illustrate our approach and validate the xUML-to-S/R translator is a robot control system[19]. Currently a simplified version of the robot control system, which is able to control a robot with one arm, has been verified. There are two joints on the arm and at the end of the arm is an end effector that moves around and performs designated functions such as grabbing. The movement of the end effector requires the two joints change their angle positions. Two major robotics algorithms are implemented in the system:

- Robot Control Algorithm
  Given a target position of the end effector, every joint calculates its target angle position. If the target angle position of every joint is less than or equal to its physical angle limit, the end effector proceeds to the target position; otherwise, a fault recovery is invoked.
- Fault Recovery Algorithm
  When a fault recovery is invoked, the angle position of the joint that violates the physical constraint is set to its physical angle limit while the other joint is required to recalculate its target angle position.

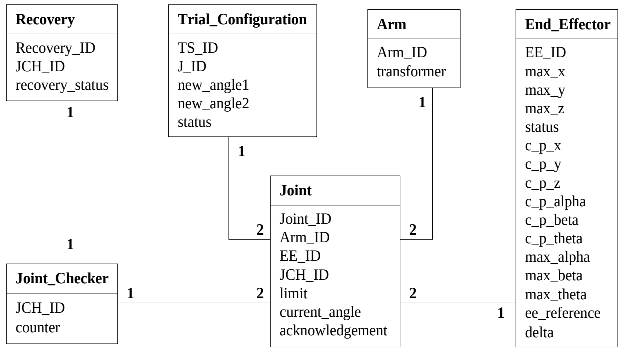

Figure 6: Class Model of Robot Control System

There are 6 classes, shown in Figure 6, in the xUML model of the simplified robot control system. Class Joint has two instances while every other class has one instance. In total, the seven class instances have 44 attributes, and 31 message types four of which have associated data items. A typical state model, the state model of the Joint class, is shown in Figure 7. It consists of 7 states and 11 state transitions. States have associated actions that can be fairly complicated.

The xUML model was automatically translated into an S/R model (not shown due to space limitation). In the S/R model, there are 15 processes, 74 state variables, and 129 selection variables in total. During the translation, 19 attributes is identified as static and translated into selection variables instead of state variables. The 74 state variables are categorized by usage as follows:

- 7 recording the current states of class instances;
- 25 modeling non-static attributes;
- 42 simulating the message queues of class instances.

42 state variables are used to encode the message queues of the class instances, which is inevitable no matter what kind of model checkable language into which the xUML model is translated because under the asynchronous message passing mechanism, the local state of each message queue contributes to the global state of the whole system.

About 20 different properties have checked against the xUML model [19]. Here we use a safety property to demonstrate how a property is defined and checked. The system design requires the robot control algorithm and fault recover algorithm work cooperatively. The safety property specifies a coordination between the two algorithms: when a fault recovery has been invoked, the second joint cannot move into the "Move_EE" state. The property is defined as follows:

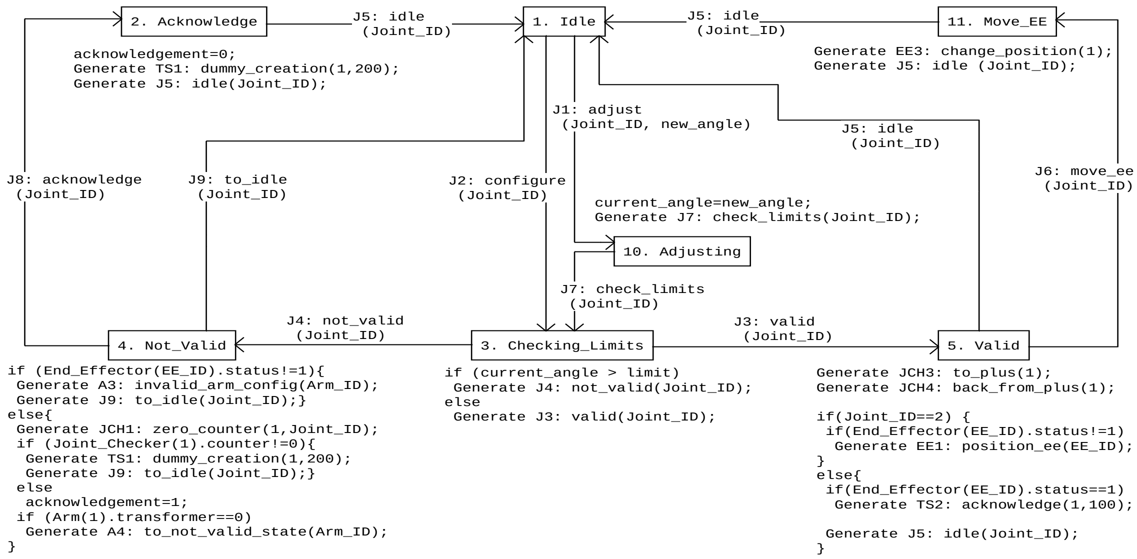

Figure 7: State Model of Joint Class

DECLARE Joint_2_in_Move_EE <<Joint 2>> $Move_EE
DECLARE Recovery_Called <<Recovery 1>>
recovery_status = 1
NEVER(Joint_2_in_Move_EE AND Recovery_Called)

The first statement defines a propositional predicate, *Joint_2_in_Move_EE*, whose value is evaluated to be true if and only if Joint 2 is in the "Move_EE" state. The second statement defines a propositional predicate, *Recovery_Called*, whose value is evaluated to be true if and only if the "recovery_status" attribute of the Recovery instance is equal to 1. The third statement declares a temporal predicate over the system: at any given moment of a system execution, *Joint_2_in_Move_EE* and *Recovery_Called* cannot be true at the same time.

The property was translated to an S/R query and checked by COSPAN against the S/R model. A failure was reported by COSPAN, which means the property does not hold of the xUML model. The error track generated by COSPAN was translated into an error report that demonstrates a system execution path in which Joint 1 invoked a recovery, but Joint 2 was not aware of the recovery and proceeded from the "Valid" state to the "Move_EE" state.

The error was corrected by a change to the action associated with the "Valid" state in the Joint state model. The change requires that Joint 2 check whether a recovery has been invoked by accessing the "recovery_status" attribute of the Recovery instance when it arrives at the "Valid" state. If a recovery has been invoked, it moves to the "Idle" state instead of the "Move_EE" state. The corrected xUML model was again translated into S/R. COSPAN checked the query against the new S/R model and reported a success, which means the property holds of the corrected xUML model.

In order to evaluate the reduction power of SPOR, both the faulty xUML model and the corrected xUML model were translated to S/R models in two different ways: SPOR off and SPOR on. The sizes and the verification complexity of the resulting S/R models are listed in Table 1. SPOR gives order of magnitude reduction on the verification complexities of both the faulty model and the corrected model. Also SPOR does not increase the number of state variables in the S/R models and only introduces a few more selection variables that do not contribute to the state space.

Although the combination of SPOR and Symbolic Verification did not lead to further reduction on the verification complexity of the simplified robot controller model, it has facilitated verification of other xUML models that cannot be verified otherwise. These results will be reported in a later paper.

7 Conclusions and Future Work

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 towards S/R models with manageable state spaces. The approach has been successfully applied to systems of modest complexity and results have been reported in [19].

The current translator handles a significant subset of xUML, but will be extended to cover the verifiable subset of xUML. Since the complexity of the resulting S/R models strongly affect the efficiency of model checking, more translation optimizations leading to S/R models with relatively small state spaces will be explored. Our approach will be applied to more complex real-world software systems to evaluate its scalability. Also the back-

Model	SPOR Off/On	No. of State Variables	No. of Selection Variables	States Searched	Running Time	Memory Usage
Faulty	Off	74	129	21490	402.6 sec	3.18M
Faulty	On	74	137	2451	47.4 sec	0.53M
Corrected	Off	74	129	55256	1054.7 sec	7.94M
Corrected	On	74	137	5081	97.7 sec	0.89M

Table 1: Model Checking Complexity Comparison

end and SPOR modules of our translator are currently being developed together with corresponding modules of SDLcheck [10], in the course of a more general project aiming at model checking support for Object-oriented technology that may involve both xUML and SDL.

8 Acknowledgments

We would like to acknowledge Robert P. Kurshan for his important role in initiating, supporting, and collaborating on this project. We would also like to thank Natasha Sharygina for providing the illustration based on the simplified robot control system and giving us valuable feedback, and Husnu Yenigun for his critical contribution to the implementation of the translator.

References

[1] Action Semantics Consortium, http://www.umlactionsemantics.org.

[2] 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.

[3] S. Gnesi, D. Latella, M. Massink, Model Checking UML Statechart Diagrams Using JACK, *Proc. of 4th IEEE International Symposium on High-Assurance Systems Engineering*, Washington DC, USA, 1999.

[4] P. Godefroid, D. Pirottin, Refining Dependencies Improves Partial-Order Verification Methods, *Proc. of 5th International Conference on Computer Aided Verification*, Springer *LNCS* no.697, 1993.

[5] R. H. Hardin, Z. Har'El, R. P. Kurshan, COSPAN, *Proc. of 8th International Conference on Computer Aided Verification*, Springer *LNCS* no. 1102, 1996.

[6] Kennedy Carter, http://www.kc.com/html/xuml.html.

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

[8] 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*, Springer *LNCS* no. 1384, 1998.

[9] G. Kwon, Rewrite Rules and Operational Semantics for Model Checking UML Statecharts, *Proc. of 3rd International Conference on UML*, York, UK, 2000.

[10] V. Levin, H. Yenigun, SDLcheck: A Model Checking Tool, to appear in the proceeding of CAV 2001.

[11] J. Lilius, I. Porres, vUML: a Tool for Verifying UML Models, *Proc. of the Automatic Software Engineering Conference*, Cocoa Beach, FL, USA, 1999.

[12] K. L. McMillan, Symbolic Model Checking, Kluwer, 1993.

[13] E. Mikk, Y. Lakhnech, M. Siegel, G. Holzmann, Implementing Statecharts in Promela/Spin. *Proc. of IEEE Workshop on Industrial Strength Formal Specification Techniques*, Boca Raton, FL USA, 1998.

[14] OMG, *OMG Unified Modeling Language Specification, Version 1.3.* OMG, 1999.

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

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

[17] D. Peled, Combining Partial Order Reductions with On-the-fly Model-Checking, *Formal Methods in System Design* no. 8, 1996.

[18] Project Technologies, http://www.projtech.com/pubs/xuml.html.

[19] N. Sharygina, R. P. Kurshan, J. C. Browne, A Formal Object-oriented Analysis for Software Reliability, *Proc. of FASE*, Genova, Italy, 2001.

[20] A. Valmari, A Stubborn Attack on State Explosion, *Prof. of 2th International Conference on Computer Aided Verification*, Springer *LNCS* no. 531, 1990.

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