Integrating Model Checking into Object-oriented Software Development Processes

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

[1] Department of Computer Sciences,
University of Texas at Austin,
Austin, TX 78712, USA.
Email: {feixie, browne}@cs.utexas.edu
[2] Bell Laboratories, Murray Hill, NJ 07974, USA.
Email: levin@research.bell-labs.com

Abstract. A methodology for integrating model checking into object-oriented software development processes is defined, developed, and demonstrated. Model checking is applied to object-oriented analysis(OOA) models that have executable semantics specified as state machines rather than as programs in conventional programming languages. The complexity level of an OOA model yields a manageable state space for model checking. An automata based approach to model checking is used. The OOA models are automatically translated to automaton models. Predicates over the behaviors of the OOA models are mapped to predicates over the automaton models and evaluated by a model checker. Algorithms for translating OOA models to automaton models are given. Procedures for management of dynamic object instance sets and unbounded event queues are given. The algorithms and procedures have been implemented for OOA models constructed in the SES/Objectbench implementation of the Shlaer-Mellor method that provides executable semantics for a subset of Unified Modeling Language. Translation is to the S/R automaton language and the COSPAN system is used for model checking. The algorithms are readily adapted to other OOA models with executable semantics and other model checking systems. A simple example to demonstrate the capabilities is included in this paper. The companion paper[7] gives design rules for constructing OOA models which yield tractable automaton models upon translation and reports on application of the methodology to an OOA model of modest complexity, a minimal robot control system.

1 Introduction and Overview

Software control of everyday systems is becoming pervasive. Software control systems are frequently complex and concurrent. Development technologies that achieve more reliable complex concurrent software systems are becoming increasingly important. This paper defines and demonstrates the integration of formal verification by model checking into object-oriented(OO) software development processes.

OO software development processes are increasingly used for the development of complex concurrent software systems, particularly embedded software systems. Figure 1 is a schematic of the conventional OO software development processes.

Most OO software development processes still largely rely on conventional testing of the source code to validate the correctness of software systems. The detection of errors in Object-oriented Analysis(OOA) models is delayed to the integration test step. Errors in OOA models caught in the integration test may force modifications in the OOA models and consequent redo of design, programming, and testing. In addition, conventional testing can never provide a complete validation of a software system due to the lack of coverage of execution paths.

Model checking [1] is a method for evaluating predicates about the behaviors of systems of concurrently executing finite state machines. Predicates are expressed as temporal logic formulas and efficient algorithms are used to evaluate the correctness of the predicates.

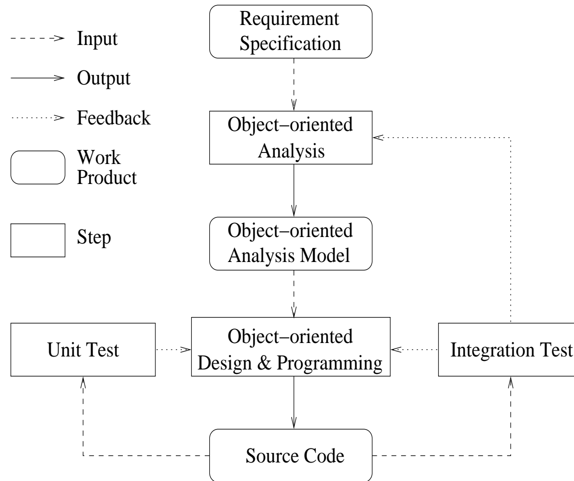

Fig. 1. OO Software Development Process

Model checking evaluates the behaviors of a system for ALL possible execution paths. Thus if a predicate is evaluated to be true under model checking, it has been rigorously verified that this predicate will hold for all executions of the system.

It is impractical to apply model checking to software systems written in conventional procedural, even OO programming languages because of the state space explosion problem. There are, however, OO software development processes, such as Shlaer-Mellor(S-M) Method [2], which express the execution behaviors of a software system as a set of concurrently executing state machines at the OOA level. The complexity level of OOA models reduces the state space explosion problem to a manageable level. This makes it conceptually possible to apply model checking to verify that the behaviors of software systems represented as OOA models conform to specifications expressed in temporal logic predicates provided that the OOA models have executable semantics.

The Object Management Group(OMG) has adopted an execution semantics for a subset of the formalizable portion of Unified Modeling Language(UML)[3]. The semantics of S-M OOA and the notation of UML define an executable subset of UML(xUML), which is consistent with the execution semantics adopted by the OMG. We are, on the recommendation of Stephen J. Mellor [private communication], referring to the OOA models developed under the S-M method as xUML OOA models.

This paper defines a methodology for integrating model checking into a commercially supported and widely used OO software development processes by application of model checking to verification of OOA models with executable semantics, describes an implementation of the methodology, and discusses applications of the methodology. In particular, model checking is applied to xUML OOA models of software systems developed using SES/Objectbench[4]. The xUML OOA models are translated to the S/R[5] automaton language and predicates against the OOA models are evaluated by the COSPAN[5] model checker, which implements the automata-theoretic approach to model checking[6]. Application of the automata-theoretic approach to model checking on xUML OOA models is practical and can be effective because:

- xUML OOA models have executable semantics.
- The execution behaviors of xUML OOA models are expressed as concurrently executing state machines.
- xUML OOA models can be (and frequently are) compiled directly to conventional programming language source code, which avoids the error-prone manual coding of OOA models.
- The automata-theoretic approach to model checking admits powerful reduction algorithms that, although only heuristics, often can break through the computational complexity barrier that impedes model checking of many problems.

The development process that integrates object-oriented development and model checking, shown in Figure 2 has the following steps:

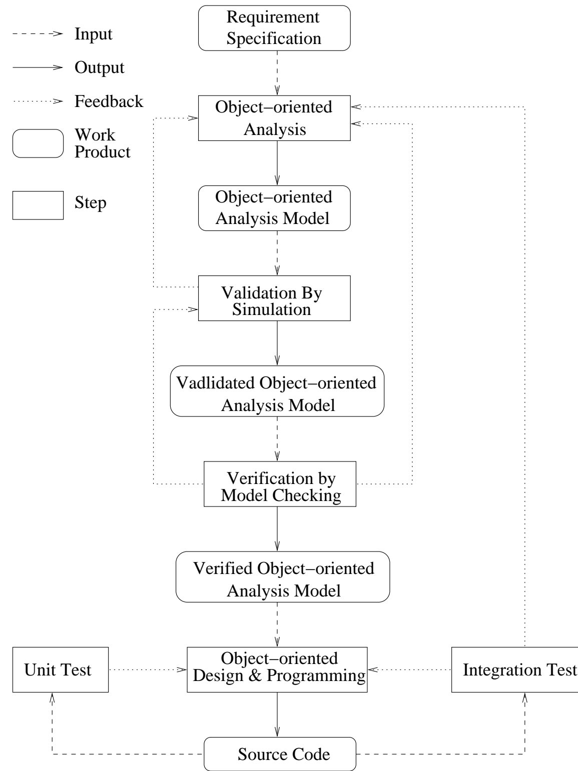

Fig. 2. Integration of Model Checking and OO Software Development Process

a. The OOA model of the system under development is obtained by analyzing the requirement specification with an OOA methodology that provides executable semantics for OOA models.
b. The OOA model is validated by execution with a discrete event simulator to obtain an OOA model that is largely correct.
c. The OOA model is fully automatically translated to an automaton model that can be checked by a model checker.
d. Predicates covering important execution behaviors of the system are specified by the designers of the OOA model.
e. These predicates is formally verified against the automaton model by model checkers. Errors found in the OOA model may result in additional validations to identify the source of the errors and/or modifications to the OOA model.
f. The steps from b. through e. are repeated until the OOA model has been verified to have the required behaviors.
g. The validated and verified OOA model is either manually programmed or more desirably, directly compiled to conventional programming language source code.

The core elements of the methodology and its implementation are:

a. Design rules for constructing OOA models to which model checking can be practically applied;
b. Algorithms for translating the semantics of executable OOA models to the semantics of the automaton models;
c. Implementation of a translator based on these algorithms;
d. Translation of predicates formulated on OOA models to predicates that can be evaluated against automaton models by model checkers.

This paper defines, describes, and illustrates elements b, c, and d. It will be shown that integrating model checking into OO software development processes by applying model checking to OOA models with executable semantics can be accomplished and is potentially scalable for application to substantial systems. Effective translations of legal but completely arbitrary OOA models is impractical. Design rules is discussed separately in [7].

2 Background

2.1 Concepts of xUML OOA Models

This section presents major concepts of xUML models.

Domains and Subsystems In building a typical large software system, analysts generally have to deal with several distinctly different subject matters, or domains. Large domains can be partitioned into subsystems. Subsystems can be viewed as clusters of classes in a domain: groups of classes that are closely interconnected with one another by relationships, but relatively few relationships connect classes of different subsystems.

Classes A class is an abstraction of a set of real-world entities such that all the entities in the set, the instances, have the same characteristics, and all instances are subject to and conform to the same model of behaviors. There are two kinds of classes:

- Active classes that have dynamic behaviors;
- Passive classes that have no dynamic behavior and are used to record information.

Every class has associated attributes. An attribute is an abstraction of a single characteristic possessed by all instances of the class. In the following discussion, we will interchangeably refer to instances of classes as “object instances”.

Lifecycles The common behavior patterns of all the instances of an active class are abstracted into a lifecycle and every lifecycle is expressed as an extended Moore state machine, that consists of:

- States. Each state represents a stage in the lifecycle of a typical instance of the class.
- Events. Each event defines a type of message that an instance of the class can receive during system executions.
- Transition Rules. A transition rule specifies which new state is realized when an instance of the class in a given state receives an instance of a particular event.
- Actions. An action is an activity or operation that must be executed when an instance arrives in a state. One action is associated with each state. An action can be composed of computations such as:
  - Calculation
  - Generation of event instances
  - Access of attributes of object instances
  - Creation or Deletion of object instances

  All computations have well defined execution semantics.

Associations An association is an abstraction of a set of relationships that hold systematically between different classes. For example, hard disks are controlled by bus controllers.

There are three forms of associations: one-to-one, one-to-many, and many-to-many. One-to-one and one-to-many associations are simply specified as the attributes of associated classes. For each many-to-many association, a separate associative class is defined, instances of which are used to record the association. The associative class can also have a lifecycle that models the dynamics of the association.

An association can involve competition between participants. When this happens, a special lifecycle, the assigner lifecycle, is built for the association. The assigner lifecycle is responsible for creating instances of the associative class to associate instances of the associated classes according to the rules of competition. There is only one instance of each assigner lifecycle during a system execution.

Generalizations When distinct specialized classes have certain characteristics in common, a more general class, the superclass, can be abstracted to represent the characteristics shared by the original specialized classes, the subclasses. Subclasses inherit the attributes and events(if defined in the superclass) of the superclass, but the subclasses do not inherit the lifecycle of the superclass. Every active subclass defines its own lifecycle.

Execution Semantics The execution semantics of OOA models is an interleaving semantics with the following properties:

- Creation and Deletion of Object Instances
  Object instances can be created either statically during the initialization of system executions, or dynamically during system executions.
  An active object instance that has a born-to-die lifecycle deletes itself when it enters a termination state. A passive object instance can be deleted in the execution of a state action of active object instances during system executions.
- Event Passing Mechanism
  Active object instances asynchronously communicate with each other by sending event instances to each other. Every active object instance has a private event queue that is FIFO and infinite. All the event instances directed to an active object instance are kept in its private event queue after being generated and before being consumed.
  Event instances in a private event queue are ordered according to their arrival order. If an active object instance generates multiple event instances to a single receiving object instance, those event instances will be enqueued in the order generated.
- Active Object Instance Scheduling
  Active object instances are scheduled for execution as follows:
  - An active object instance is ready to be scheduled if either it has entered its current state and is ready to execute the associated action of the state, or it has finished the action of its current state and is ready to perform a state transition by consuming an event instance and there are event instances in its private event queue.
  - At any given moment of a system execution, exactly one active object instance is scheduled nondeterministically to execute among all ready active object instances.
  - The scheduled active object instance either performs a state transition by consuming an event instance in its private event queue, which changes its state, or executes a state action, which can create or delete object instances, send event instances to other active object instances, access its own attributes, or access the attributes of other object instances.
  - Both the execution of a state action and the performance of a state transition are run-to-completion.

2.2 Automata-theoretic Approach to Model Checking

This section gives a brief introduction to the automata-theoretic approach to model checking.

ω-automata and Their Languages ω-automata are the same as conventional automata that accept strings, except that the final states of the latter(signaling the end of an accepted string) are replaced by an acceptance condition on the set of states visited infinitely often. The language of ω-automaton A, $\mathcal{L}(A)$, is defined to be the set of all infinite sequences accepted by A.

Approach Overview The automata-theoretic approach to model checking is founded on L-ω-automata, a class of ω-automata defined in [6]. The system is modeled by an L-ω-automaton P, where P represented as a synchronous parallel composition $P = P_1 \otimes P_2 \otimes \ldots \otimes P_k$ of component processes(all modeled as L-ω-automata). The property to be checked is represented by the 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.

In the following discussion, automata refers to L-ω-automata and automaton models refer to L-ω-automaton models.

Automata as Processes In the S/R automaton language, an automaton is represented as a process. A system is composed of a set of processes that are synchronously interacting with each other.

A process consists of

- State Variables. The current values of all state variables define the current state of the process. The state space of the process is determined by the ranges of all state variables.
- Selection Variables. Selection variables define the selections, outputs of the process, corresponding to each state. Selections differ from conventional state machine outputs in that the latter are assumed to be a deterministic function of the states, whereas selections are nondeterministic outputs, the ranges of which are a function of the states.
- Inputs. Each process chooses a subset of all the selection variables of other processes in the system as its inputs.
- State Transition Predicates. State transition predicates specify how the process changes its state by updating its state variables as a function of its current state and inputs.
- Selection Rules. Selection rules assign values to selection variables as functions of state variables, other selection variables, and inputs.

The Selection/Resolution Model The system execution model, defined in the automata-theoretic approach to model checking, namely the "selection/resolution" model is a clock-driven synchronous execution model, under which a system of processes behaves in a two-step procedure every clock cycle as shown in Figure 3:

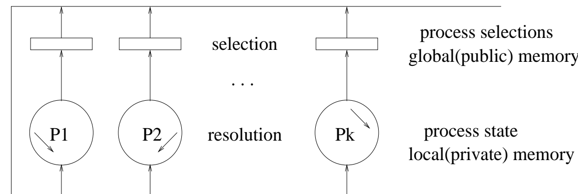

Fig. 3. Selection/Resolution Model

- Each process "selects" a selection possible from 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.
- Each process "resolves" the current global selection by moving to a new state through a transition whose associated predicate is enabled by the current global selection.

2.3 Symbolic Verification

Symbolic verification is based on the symbolic search of the model's state transition graph [8]. At each step of the symbolic search, the reached set of states is symbolically represented as

a whole by a boolean function and the latter is stored as a Binary Decision Diagram [9]. Since symbolic search can manipulate a set of states at a time, it is capable of searching exponentially larger state spaces than is possible with explicit search. Thus, symbolic verification offers, at least, potentially a relief of the state space exponential explosion that is the main obstacle in incorporation of model checking into software development. It is the reason that motivates our choice of the automata-theoretic approach of model checking, which intrinsically allows symbolic verification. However, with this choice we get a synchronous automaton semantics and, hence, a non-trivial task of translation from the asynchronous software semantics into the synchronous automaton semantics, which is discussed next.

3 Translation of OOA Models to Automaton Models

The translation is from OOA models with an asynchronous interleaving semantics of execution, dynamic creation and deletion of object instances, and potentially unbounded state spaces to automaton models with a synchronous parallel semantics of execution, a static set of interacting automata, and finite state spaces. The translation must not only map across these disparate semantic domains, but also ensure that the state spaces of the resulting automaton models are not only finite but also of manageable size. The algorithms given in this section enable demonstration of effective translation and model checking of interesting OOA models but there are significant open problems and much further work on translation algorithms is needed. Some automated transformation of OOA models is also done by the translator prior to translation to automata models.

3.1 Modeling Asynchrony with Synchrony

The execution semantics of OOA models is intrinsically asynchronous while automaton models have an inherently synchronous execution semantics. To translate OOA models to automaton models, we simulate the asynchronous execution semantics of OOA models with the synchronous execution semantics of automaton models.

Modeling Asynchronous Event Passing The event passing mechanism employed by OOA models is asynchronous. Suppose I_1 and I_2 are two active object instances, defined in the OOA model of a particular software system, and they communicate with each other by event passing. When I_1 sends an event instance, e, to I_2, e is put into the private event queue of I_2 when it is generated by I_1. No acknowledgment to e is required from I_2 before I_1 can proceed with other processing. I_2 is not aware of the arrival of e until it is ready for consuming an event instance and e is the first event instance in its private event queue.

The execution semantics of the automata model specifies synchronous communication. Each automaton posts its selection variables for a clock cycle and each automata inputs selection variables of other automata in the same clock cycle

The asynchronous event passing mechanism of OOA models can be simulated by the synchronous communication mechanism of automaton models through modeling the private event queue of every active object instance as an automaton. Suppose automata IA_1 and IA_2 model two active object instances and automata QA_1 and QA_2 model their corresponding private event queues. An event instance, e, can be sent from IA_1 to IA_2 by the following steps:

- $[IA_1 \rightarrow QA_2]$ IA_1 passes e to QA_2 through synchronous communication.
- [Buffered] QA_2 keeps e until IA_2 is ready for consuming a event instance and e is the event instance at the head of the queue modeled by QA_2.
- $[QA_2 \rightarrow IA_2]$ IA_2 receives e from QA_2 through synchronous communication.

There is no synchronization between IA_1 and IA_2. QA_2 may keep several event instances for IA_2 at any given moment of a system execution. After the first event instance in the queue is received by IA_2, QA_2 deletes the first event instance by moving the other event instances in the queue one slot toward the head of the queue simultaneously.

Modeling Asynchronous Execution Under the execution semantics of OOA models, only one active object instance can execute a state action or perform a state transition at a given moment while under the execution semantics of automaton models, in any given clock cycle, every automaton resolves the current global selection by making a state transition simultaneously.

The asynchronous execution of active object instances can be simulated by the synchronous execution of automata as follows:

– Every automaton modeling an active object instance has a selection variable, the ready indicator, which indicates whether the active object instance modeled is ready for executing an OOA state action, or performing an OOA state transition.
– A global scheduler, also modeled as an automaton, inputs the ready indicators of all the automata modeling active object instances. When a rescheduling occurs, the global scheduler nondeterministically selects one active object instance among all the active object instances that are ready. The global scheduler has a selection variable, the grant indicator, which indicates which active object instance is selected at a given moment.
– All the automata modeling active object instances input the grant indicator. Only the automaton that models the selected active object instance can perform an automaton state transition corresponding to an OOA state action or an OOA state transition in the lifecycle of the selected active object instance. All other automata modeling active object instances follow a self-loop automaton state transition back to their current automaton states.

3.2 Model Checking Optimization via Model Transformation

State space reduction is a critical problem for scalable application of model checking. Different algorithms for state space reduction are applicable in each of the several modes of model checking. Symbolic verification is readily applied to synchronous automata while partial order reduction [10] [11] [12] is readily applied to asynchronous interleaving automata

Partial order reduction follows the observation that in many cases, when components of a system are not tightly coupled, different orders of execution of two or more transitions that belong to different components may result in the same global state. Then, under some conditions (in particular, when the interim global states are not relevant to the property being checked), entirely formulated in [10] [11] [12], model checking may explore just along one of the possible execution orders, yet preserving the property.

The asynchronous interleaving semantics of the OOA models suggests implementation of a static partial order reduction on the OOA models prior to the translation to synchronous automata, namely, as *model transformation*. This transformation follows the procedure specified in [13]. This enables integrated application of the static partial order reduction algorithms when applying symbolic verification to the OOA models using synchronous automata.

3.3 Mapping Object Instances to Automata

A passive object instance is translated to a single automaton while an active object instance is mapped to a pair of automata, one modeling its lifecycle and the other modeling its private event queue.

Translation of Attributes Attributes of a passive object instance are translated to the state variables of the corresponding automaton while attributes of an active object instance are translated to the state variables of the automaton that models its lifecycle.

In OOA models, the attributes of an object instance may be accessed by other object instances. Read accesses are implemented through variable inputing between automata while since state variables of automata cannot be directly updated by other automata, write accesses are simulated as follows:

– If an attribute of an object instance, the owner, is write accessed by other object instances, the writers, during system executions, in every writer's corresponding automaton an additional selection variable is defined, which is assigned the value for the attribute when the write access is executed.
– All these selection variables are inputted by the owner's corresponding automaton. The state variable corresponding to the attribute in the owner automaton is updated according to the value of the selection variables inputted when the writer automata execute write accesses.

There are some attributes of classes, whose values never change during system executions. Such attributes are translated into selection variables that do not contribute into the state space, instead of state variables. This is a crucially important optimization for model checking because it leads to an automaton model with a smaller state space.

Translation of Lifecycles The lifecycle of an active object instance is translated into the transition structure of the corresponding automaton as follows:

– Every OOA state in the lifecycle is translated into an automaton state, namely the transition state.
– Every OOA action is translated into a set of automaton states, namely action states. Every action state has associated transition predicates leading to its successive action states in the set or the corresponding transition state.
– Every transition rule is translated into a transition predicate associated with the corresponding transition state and leading to one of the action states corresponding to the target OOA state of the OOA transition in the lifecycle.
– An event labeling a transition rule is translated into a condition expression that checks whether the first event instances kept in the automaton modeling the private queue of the object instance is of the event required.

3.4 Translating Associations

An association relationship is entirely defined by the attributes and lifecycles of the two associated classes and the associative class(if the association is many-to-many), and the assigner lifecycle(if the association models competition). Translation of an association relationship is reduced to the translation of the object instances involved in the association and the translation of the assigner lifecycle. All the involved object instances are translated as discussed above and every assigner lifecycle is modeled by a pair of automata, one modeling the lifecycle and the other modeling its private event queue.

3.5 Translation Generalizations

In the translation of a generalization, the attributes defined in the superclass and inherited by the subclasses are treated the same as the attributes defined in the subclasses.

The core execution behavior conveyed by a subclass/superclass relationship among active classes is the dynamic dispatching of instances of superclass events(events defined in the superclass), to instances of the subclasses. The translator deals with the dynamic dispatching as follows:

– In every automaton corresponding to an instance of a subclass, for every superclass event, a corresponding subclass event is defined. Every appearance of the superclass event is replaced by the corresponding subclass event.
– In the translation of every state action in which a superclass event instance is generated to a subclass instance, a dynamic mapping mechanism is inserted, which converts the superclass event instance to an instance of the corresponding subclass event according to which subclass the target object instance belongs to and the resulting subclass event instance is sent to the subclass instance.

3.6 Guaranteeing Fixed Finite State Spaces

Model checkers require that the models that are passed to them for checking have finite state spaces and these finite state spaces cannot be expanded and contracted through dynamic creation and deletion of components of the models.

Ranging Data Types A finite range must be provided for each integer type and each enumeration type. A continuous infinite type, like the float type, must be discretized and represented by an integer interval type or an enumeration type. All these are reasonable requirements posted to the designers of software systems because no computer system supports integer types of infinite range and float types with infinite precision.

Ranging data types can also help detect out of range errors since if a variable is assigned a value out of range, it will be detected by the model checker.

Simulating Dynamic Creation and Deletion of Object Instances If instances of a class O, can be dynamically created and deleted during system executions, the dynamic creation and deletion is simulated in automaton models as follows:

- The upper bound, N, of the number of instances of O that can exist at the same time during system executions, is estimated.
- The translator generates N automata, $A[0] \ldots A[N-1]$, and each $A[i]$, $0 \leq i \leq N-1$, models an instance of O.
- In every $A[i]$, $0 \leq i \leq N-1$, an additional state variable, *liveness*, is added to indicate whether the automaton is currently representing an existing instance of O.
- When an instance of O is created, $A[j]$, $0 \leq j \leq N-1$, whose *liveness* is false, is selected and its *liveness* is set to be true. Then it participates in the system execution by interacting with other automata.
- When an instance of O is deleted, *liveness* of the corresponding $A[k]$, $0 \leq k \leq N-1$, is set to be false and $A[k]$ stops interacting with other automata.

Managing Event Queue Overflow Under the semantics of OOA models, every active object instance has an associated private event queue that is assumed to have an infinite number of slots. In the automaton models, every private event queue is modeled by an automaton that clearly cannot model an event queue that has an infinite number of slots. This opens the possibility of event queue overflow that may affect verification results. We currently deal with event queue overflow as follows:

- An upper bound of the number of slots needed during system executions is heuristically estimated for every event queue by the designer of the OOA models. How to automatically estimate these upper bounds based on the OOA models of the system is an open problem.
- The automata modeling the private event queues are constructed based on the upper bounds estimated.
- When the verification of a query reports false, an error track processing tool is used to analyze for actions becoming blocked by attempting to place a generated event instance into a full queue. When there is such a blockage, the verification will be invalidated and should be redone with a larger size for the queue that caused the blockage.

4 Implementation Issues

This section discusses some implementation issues. In the following discussion, we refer to the translator that translates OOA models to automaton models as OOA-to-Automaton translator.

4.1 Textualization of Graphical OOA Models

OOA models are represented graphically. It is not suitable to use the graphical OOA models directly as the input to the OOA-to-Automaton translator.

We have defined a textual representation of OOA models that

- Is readily parsed;
- Correctly captures all the information that graphical OOA models convey.

We have implemented a preprocessor for the OOA-to-automaton translator that takes graphical OOA models as input, and textualizes them to generate textual OOA models.

4.2 Implementation of OOA-to-Automaton Translator

The current implementation of the OOA-to-Automaton Translator is based upon extending a SDL-to-S/R translator [13] [14] that is currently being developed by Levin, et.al. The SDL-to-S/R translator first translates a SDL program to the abstract syntax tree(AST) for the SDL program and the AST is then translated to an automaton model in the S/R automaton language. It requires two extensions to the SDL-to-S/R translator to implement the transition from OOA models to automaton models. The AST for SDL programs was extended to cover the OOA model semantics and the SDL-to-S/R translator was extended to cover the OOA model semantics now represented in the extended SDL AST. An OOA model is translated to its corresponding automaton model using the sequence of transformations shown in Figure 4:

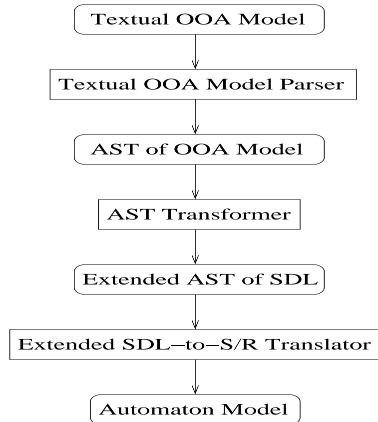

Fig. 4. Sequence of Model Transformations

- The Textual OOA Model Parser takes the textual OOA model of a system as input and outputs the AST of the OOA model.
- The AST Transformer inputs the AST of the OOA model and transforms it to the corresponding extended SDL AST.
- The Extended SDL-to-S/R Translator inputs the resulting extended SDL AST and generates the corresponding automaton model.

4.3 Analysis Tools and Support

Effective use of model checking requires that the inputs to the model checking process be readily formulated and outputs of the model checking process be readily understood by the software system designers. A query translator and a post-processor for error tracks are provided to enable effective application of model checking to OOA models.

Query Specification Queries to be checked for an OOA model must be expressed as queries referring to the translation of the OOA model to the automata model. Effective application of model checking to the OOA model requires that the queries be expressed in the OOA model representation. An interface for specification of queries in the OOA model representation is provided along with a translator from the OOA-specified queries to the automata-specified queries.

Post-Processing of Error Tracks When a query fails, the model checker generates an error track specifying an execution trace of the automaton model that is inconsistent with the query. A translator is provided, which maps the error track to a representation in terms of the OOA model representation.

5 Illustration of Methodology

The real-world application we have been using to test the design and implementation of the model translating algorithms and the translator is a robotic control system. Space limitations preclude using the robotic control system as the illustration. A detailed specification of the robotic control system can be found in [7]. In this section, the dining philosophers problem is used to illustrate our methodology.

A solution to the problem is designed by applying the S-M method. The OOA model consists of two classes, philosopher, whose lifecycle is shown in Figure 5, and chopstick,

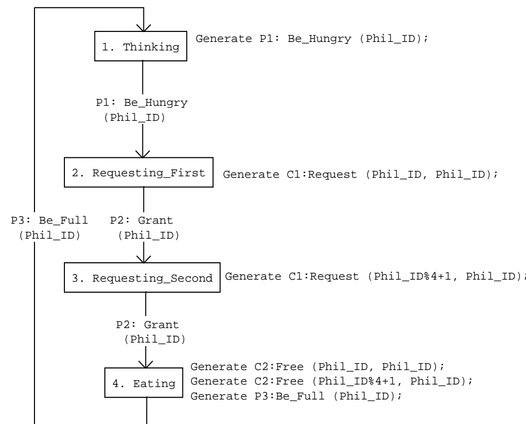

Fig. 5. Lifecycle of Philosopher

whose lifecycle is shown in Figure 6. Both philosopher and chopstick have four instances. Each instance of philosopher and chopstick has a unique integer as its Id, ranged in $1 \ldots 4$ and recorded in the attributes, *Phil_ID* and *Chop_ID* respectively. Sending of event instances is specified as generate statements in lifecycles. For example, in the state of *Thinking*, a philosopher sends itself an instance of event *P1* by *Generate P1: Be_Hungry(Phil_ID)*. A philosopher attempts to pick up first her left chopstick and then her right chopstick when she is hungry.

The designer can check whether the system is free of deadlock and starvation by the following query:

```
DECLARE THINKING <<Philosopher 1>> $Thinking
DECLARE HUNGRY <<Philosopher 1>> $Requesting_First
DECLARE EATING <<Philosopher 1>> $Eating
AfterEventually(HUNGRY, EATING)
AfterEventually(EATING, THINKING)
```

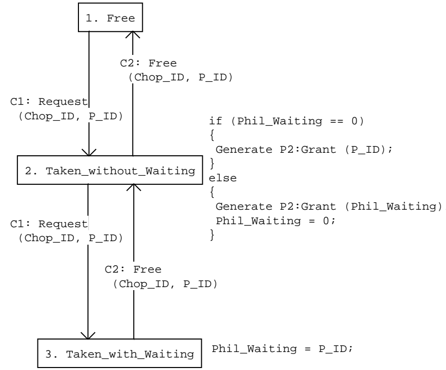

Fig. 6. Lifecycle of Chopstick

The first statement in the query defines a propositional predicate, *THINKING*, whose value is evaluated to be true if and only if *Philosopher 1* is in the state of *Thinking*; The second statement defines a propositional predicate, *HUNGRY*, whose value is evaluated to be true if and only if *Philosopher 1* is in the state of *Requesting_First*; The third statement defines a propositional predicate, *EATING*, whose value is evaluated to be true if and only if *Philosopher 1* is in the state of *Eating*; The fourth statement declares a temporal predicate over the system: If at a given moment of the system execution, *HUNGRY* is true, then eventually at a following moment, *EATING* must be true; The fifth statement declares a temporal predicate: If at a given moment of the system execution, *EATING* is true, then eventually at a following moment, *THINKING* must be true. Since philosophers are symmetric to one another, only one philosopher must be checked.

To verify the property specified by the query, both the OOA model and the query are translated into automaton models by our automatic model translator. The resulted automaton models are verified by COSPAN.

The verification reports that the first AfterEventually predicate is not satisfied and an error track is generated by COSPAN, which shows a deadlock that all philosophers are stuck in the state of *Requesting_Second*, which means they all get one chopstick and request the second one, but no chopstick is available, thus the system cannot progress.

The deadlock can be avoided as follows. An "odd" philosopher that has an odd integer as its unique Id, picks up her left chopstick first and then her right chopstick, whereas an "even" philosopher picks up her right chopstick first and then her left chopstick. The corrected lifecycle of philosopher is shown in Figure 7.

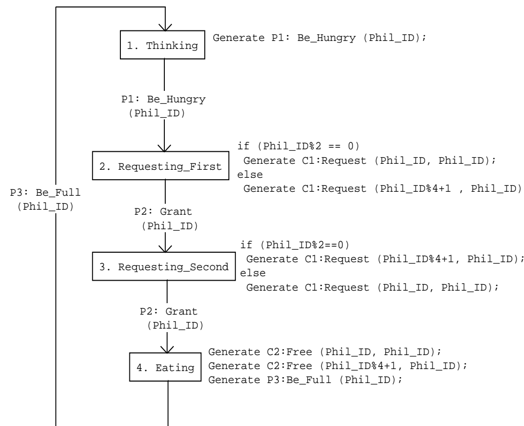

Fig. 7. Corrected Lifecycle of Philosopher

The translation and verification steps are repeated. The verification reports a success that means the corrected system has the property specified by the query.

6 Related Work

Previous work on application of model checking to software systems has mainly been either to software systems written in procedural languages or to abstract models extracted from programs in procedural languages. Feaver[15] targets software systems written in C while [14], [16], and [17] focus on applying model checking on SDL programs. Havelund and Pressburger[18] apply model checking to Java programs. Corbett, et.al[19] extract finite state machines from Java programs to which model checking is applied.

The closest research to this project is the vUML tool described in [20]. The vUML tool translates a subset of UML into Promela[21] and applies SPIN[21] for model checking. However, the cited work does not consider the overall OOA methodology enriched with model checking (cf. Figure 2). The subset of UML which is translated is incompletely specified. The algorithms for translation are rather simplistic. No model checking optimizations are suggested, except those which are already embedded in SPIN. State machine actions are coded in the procedural language of Promela.

This project integrates formal verification by model checking into a commercially supported and widely applied object-oriented development process. Verification through model checking is applied at the OOA level. This approach not only has the advantage that OOA models, being more abstract, lead to automata models which have smaller state spaces than programs in conventional languages but it also enables detection and correction of analysis and design errors prior to implementation.

Most previous application of model checking tools to software systems make use of model checkers based on interleaving automaton models[15], [16], [17], and [20]. We deliberately choose the COSPAN model checker based on synchronous automaton models to enable application of both symbolic verification, localized reduction and partial order reduction with the goal of application to complex systems.

There are also several projects which apply model checking to software developed in experimental language systems. Quest[22] for Autofocus[23] is one such example.

7 Conclusion and Future Work

This paper defines, demonstrates, and illustrates a methodology for integrating model checking into Object-oriented software development processes through automatic model translation. The issues of application of model checking to the formal verification of OOA models with executable semantics have been defined and a framework for resolution described. A feasibility demonstration implementation based on translation of xUML OOA models as implemented in SES/Objectbench to the S/R automata language and use of the COSPAN model checking system has been accomplished. Model checking has been successfully applied to systems of modest size and issues of scalability explored.

Future work will be devoted to the following topics:

- Translation Optimizations
  The complexity of the automaton models strongly affect the efficiency of model checking. Translation optimizations that can lead to automaton models with relatively small state spaces need to be explored.
- Timing Verification
  S-M method also supports the analysis of timed behaviors of complex concurrent software systems. The automata-theoretic approach to model checking is effective at timing verification. Support to timing verification in the automatic model translation will be explored.
- Applications to More Complex Systems
  Our methodology will be applied to more complex real-world software systems to demonstrate its scalability.

8 Acknowledgments

We gratefully acknowledge Robert P. Kurshan for his important role in initiating, supporting, and collaborating on this project. Natasha Sharygina's valuable feedback and Hüsnü Yenigün's contribution were both critical to the implementation of the translator. Stephen J. Mellor made valuable suggestions for the content of this paper.

References

1. E. M. Clarke, E. A. Emerson, Design and Verification of Synchronization Skeletons for Branching Time Temporal Logic, *Proc. of Logic of Programs Workshop*, Yorktown Heights, NY, USA, Springer *LNCS* no. 131, pp. 52-71, 1982.
2. S. Shlaer, S. J. Mellor, *Object Lifecycles Modeling the World in States*, Prentice-Hall Inc., 1992.
3. Object Management Group(OMG), *Action Semantics for the UML*, OMG, 2000.
4. SES Inc, *Objectbench Reference Manual*, SES Inc., 1998.
5. R. H. Hardin, Z. Har'El, R. P. Kurshan, COSPAN, In *Proc. of 8th International Conference on Computer Aided Verification*, Springer *LNCS* no. 1102, pp. 423-427, 1996.
6. R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, Princeton University Press, 1994.
7. N. Sharygina, R. P. Kurshan, J. C. Browne, A Formal Object-oriented Analysis for Software Reliability, Submitted to FASE 2001.
8. K. L. McMillan, *Symbolic Model Checking*, Kluwer, 1993.
9. R. E. Bryant, Graph Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers* C-35, pp 677-691, 1986.
10. A. Valmari, A Stubborn Attack on State Explosion, *Prof. of 2th International Conference on Computer Aided Verification*, Springer *LNCS* no. 531, pp. 156–165, 1990.
11. P. Godefroid, D. Pirottin, Refining Dependencies Improves Partial-Order Verification Methods, *Proc. of 5th International Conference on Computer Aided Verification*, Springer *LNCS* no.697, pp. 438–449, 1993.
12. D. Peled, Combining Partial Order Reductions with On-the-fly Model-Checking, *Formal Methods in System Design* no. 8, 39–64, 1996.
13. 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, pp. 345-357, 1998.
14. E. Bounimova, V. Levin, O. Başbuğoglu, and K. İnan, A Verification Engine for SDL Specification of Communication Protocols, *Proc. of the First Symposium on Computer Networks*, Istanbul, Turkey, pp. 16-25, 1996.
15. G. J. Holzmann and M. H. Smith, Feaver: Automating software feature verification, Bell Labs Technical Journal, Vol. 5, 2, pp. 72-87, 2000.
16. M. Bozga, J. C. Fernandez, L. Ghirvu, S. Graf, J. P. Krimm, L. Mounier, J. Sifakis, IF: An Intermediate Representation for SDL and its Applications. *Proc. of the SDL Forum*, Montreal, Canada, 1999.
17. D. Bosnacki, D. Damm, L. Holenderski, N. Sidorova, Model checking SDL with Spin, *Proc. of the Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, 2000.
18. K. Havelund, and T. Pressburger, Model Checking Java Programs Using Java PathFinder, International Journal on Software Tools for Technology Transfer (STTT) 2(4), 2000.
19. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Bandera: Extracting finite-state models for Java source code, *Proc. of 22nd ICSE*, 2000.
20. J. Lilius, I. Porres, vUML: a Tool for Verifying UML Models, *Proc. of the Automatic Software Engineering Conference*, Cocoa Beach, FL, USA, 1999.
21. G. J. Holzmann, The Model Checker Spin, *IEEE Trans. on Software Engineering* Vol. 23, No. 5, 1997.
22. Oscar Slotosch, Overview over the project Quest, Springer *LNCS* no. 1641, pp. 346-350, 1999.
23. F. Huber, B. Schätz, A. Schmidt, K. Spies, AutoFocus - A Tool for Distributed Systems Specification, *Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems*, Springer *LNCS* no. 1135, 1996.