Integrated State Space Reduction for Model Checking Executable Object-oriented Software System Designs

Fei Xie and James C. Browne

Department of Computer Sciences,
University of Texas at Austin,
Austin, TX 78712, USA.
Email: {feixie, browne}@cs.utexas.edu
Fax: +1 (512) 471-8885

Abstract. This paper presents a general framework for integrated state space reduction in model checking executable object-oriented software system designs. The framework structures the application of state space reduction algorithms into three phases with different algorithms applied in each phase. The interactions between these algorithms are explored to maximize the aggregate effect of state space reduction. Automaton support for the framework has been proposed and partially implemented. The framework is presented for system designs modeled in xUML [1][2], an executable subset of UML, but can also be used to structure integrated state space reduction for other representations. To further improve the applicability of the framework, domain-specific design patterns can be explored to instantiate the framework for different application domains. An instantiation of the framework for distributed transaction systems is defined and its partial implementation has been applied to the design model of an online ticket sale system. The dimension of the software system designs that are model checkable is found to be greatly extended.

1 Introduction

Executable object-oriented modeling languages such as xUML [1][2], an executable subset of UML, are widely applied in industry to model software system designs. Model checking [3][4] can potentially enhance the reliability and robustness of executable object-oriented software system designs. However, model checking software system designs of arbitrary size is intractable due to the well-known state space explosion problem. Therefore, state space reduction algorithms have to be applied to reduce the model checking complexity.

This paper defines and describes a general framework for integrated state space reduction in model checking executable object-oriented software system designs. The framework assumes that the executable system designs can be translated into model checkable languages and is discussed using system designs modeled in xUML. An earlier paper [5] has been focused on model checking an xUML model by translating the model into the S/R [6] automaton language. Under the framework, state space reduction algorithms can be applied to xUML models before translation and during translation, and applied on the resulting S/R model. Interactions between these algorithms are explored to maximize the aggregate effect of state space reduction.

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

Distributed transaction systems, which are commonly constructed in a design pattern of dispatchers, agents, and servers with customer initiated transactions as observable units of work, are examples of a family of systems for which a structured process for application of state space reduction algorithms at the design model level can be formulated. This paper illustrates the framework with its instantiation for distributed transaction systems: a systematic process for reducing model checking a property on the model of a transaction system into checking a well-defined set of less complex model checking problems. The process represents a transaction as message sequences, associates the property to be checked with a transaction, partitions the model into sub-models, and decomposes the property into sub-properties and assumptions defined over these sub-models. The process is evaluated by its application in model checking an online ticket sale system. The dimension of transaction systems that can be model checked is materially extended by the systematic process.

There has been extensive research on state space reduction algorithms for either hardware systems or software systems, which is surveyed in [4]. Our work, instead of focusing on particular state space reduction algorithms, explores the integrated application of reduction algorithms in the context of the general framework and investigates how domain specific design patterns can help adapt the general framework to different application domains to achieve better state space reduction. Our work is distinguished from the integrated state space reduction for hardware systems [7][8] by focusing on software systems and incorporating both reduction algorithms effective for asynchronous semantics and those effective for synchronous semantics.

Section 2 defines the general framework, informally describes the state space reduction algorithms currently applied in the context of the framework, and gives some guidelines for when to apply each state space reduction algorithm and the application order of these algorithms. Section 3 sketches the partially implemented automation support for the general framework. Section 4 defines, describes, and illustrates the instantiation of the general framework on distributed transaction systems. Section 5 evaluates the instantiation with results from model checking the online ticket sale system. Section 6 gives the conclusion.

2 Integrated State Space Reduction

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

2.1 General Framework

The model checking process for an xUML model, previously reported in [5], is a sequential application of the following two procedures on the xUML model:

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

The process, referred to as the basic model checking process in Figure 1, works effectively on xUML models with small numbers of class instances, but cannot scale due to the state space explosion problem. On the other hand, for

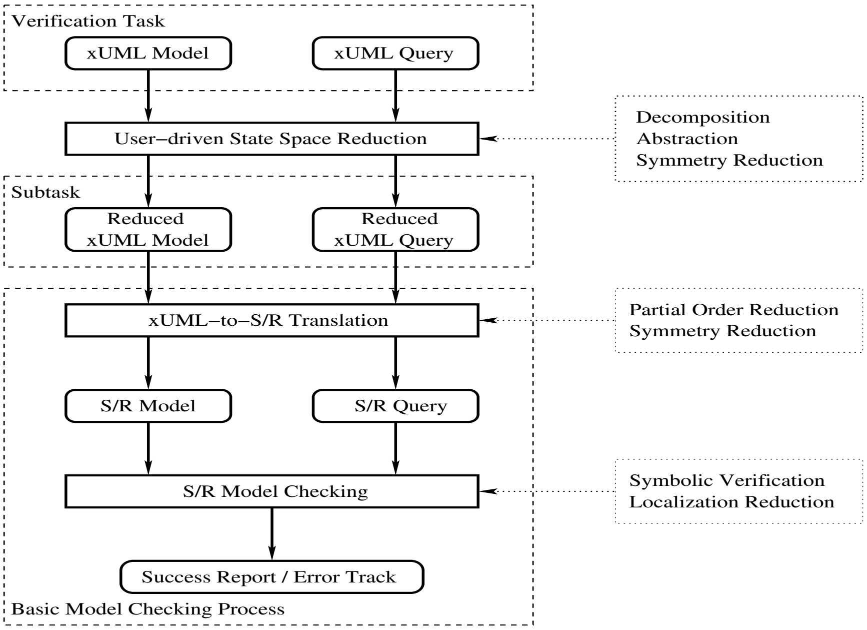

Fig. 1. Reduction Hierarchy of General Framework

well-structured xUML models, there are system structure specific and property specific reduction algorithms at the xUML model level, which cannot be recognized by the xUML-to-S/R translator and the COSPAN model checker, but which can effect major state space reduction on the resulting S/R model that is to be model checked. Therefore, the general framework prefaces the basic model checking process with a user-driven state reduction procedure.

The general framework establishes a three-level hierarchy for integrated state space reduction, as shown in Figure 1. Different reduction algorithms are invoked on different levels of the hierarchy and applied on models of different forms:

- In the user-driven state space reduction procedure, user-driven reduction algorithms such as decomposition, abstraction, and symmetry reduction are applied to reduce a complex model checking task T, a complex query on a complex xUML model, into a set of subtasks. Each subtask checks a sub-query of the original query on a sub-model of the original model. A sub-model is either a component or an abstraction of the original model. Each subtask is either discharged by invoking the basic model checking process or further reduced. The reductions applied are validated by invoking the basic model checking process or conducting a simple theorem proving.
- In the xUML-to-S/R translation procedure, automatic reduction algorithms, such as static partial order reduction and symmetry reduction, are applied. These algorithms transform an xUML model prior to its translation into S/R with respect to a given xUML query and construct an equivalent xUML model that has a smaller state space.
- In the S/R level model checking procedure, automatic reduction algorithms implemented by COSPAN, such as symbolic model checking and localization reduction, are applied. These algorithms make use of the semantic information of an S/R model to reduce the state space to be explored by COSPAN.

Under the general framework, the extended model checking process for xUML models operates recursively and interactively as shown in Figure 2. A model

```
Enqueue(ToDo, T0); Done ={ }; /* ToDo is a queue and Done is a set. */
Do
  T = Dequeue(ToDo);
  If (T is Directly Model Checkable) Then
    If (Basic-model-checking-process(T)) Then
      Done = Done + {T}; Continue;
    Else
      Error-report-generation(T); Invoke-user-interface ( );
  End;
  < T1, ..., Tn > = User-driven-state-space-reduction(T);
  If (Valid(T, < T1, ..., Tn >) Then Enqueue(ToDo, T1, ..., Tn);
  Else
    Error-report-generation(T, < T1, ..., Tn >); Invoke-user-interface( );
  End;
Until (Empty(ToDo));
```

Fig. 2. Recursive and Interative Model Checking Process Under General Framework

checking task, T_0, is recursively reduced into subtasks. A reduction conjecture from users is always validated before its resulting model checking subtasks are discharged. The basic model checking process becomes a model checking engine for discharging subtasks. When a reduction conjecture or a subtask is verified to be false, user intervention is requested. Upon user inputs, either the original property is concluded to be false or a new reduction conjecture is introduced.

2.2 Major State Space Reduction Algorithms

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

Decomposition The compositional hierarchy and asynchronous message communication semantics of xUML make decomposition a natural state space reduction algorithm for xUML models. All the three decomposition algorithms mentioned below can also be categorized as abstraction algorithms.

- **Query Decomposition** A complex query on an xUML model can be decomposed into a set of sub-queries on the original model, its components, or its abstractions. The conjunction of these sub-queries implies the original query. Every sub-query is then checked separately.
- **Component-based Decomposition** To facilitate the query decomposition, the hierarchical structure of an xUML model is explored to decompose the model into components that have simple and clear interfaces between each other. Dependencies between components are formulated as assumptions of every component on other components. Therefore, a sub-query can be checked on a component under its assumptions, which consumes less memory and time than checking the sub-query on the original model.
- **Case Splitting** In many xUML models, concurrent operations may be grouped into units of work, for example transactions in an e-business system. Commonly there is little interaction between these units of work. If a query on the whole system can be decomposed into sub-queries on units of work and the units of work can be decoupled when these sub-queries are checked, significant state space reduction can often be achieved.

Abstraction Three additional abstraction algorithms can be applied:

- **State Model Abstraction** An xUML model may be abstracted by replacing a set of state models with a set of simpler state models so that the execution behaviors of the abstract model are a super set of the execution behaviors of the original model. Therefore, if a query to be checked on the original model is verified to be true on the abstract model, we can then conclude that the query is also true on the original model.
  The most common form of state model abstraction is the non-deterministic abstraction. For instance, a decision point in a state model can be made non-deterministic and a set of state models with unique identities can be simulated by a state model with a non-deterministic identity. A major advantage of non-deterministic abstraction over other kinds of state model abstraction is that its correctness is automatically guaranteed.

- **Data Abstraction** If a mapping can be found between data values of an xUML model and a small set of abstract data values, then an abstract xUML model can be constructed by extending the mapping to states and transitions. Since the state space of the abstract model is often smaller, it is frequently easier to check properties on the abstract model.
- **Localization Reduction** Given a model and a property, localization reduction [9], also known as cone of influence reduction [4], eliminates variables in the model that do not influence the variables in the property. The checked property is preserved, but the size of the model to be checked is smaller.

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

- **Symmetric Query Reduction** Given two queries on an xUML model, if a nontrivial mapping can be defined between variables in the model or between values of variables, which maps the model to itself and the two queries to each other, then only one of the two queries need to be checked on the model.
- **Quotient Model Reduction** Having symmetry in a model implies the existence of nontrivial permutation groups that preserve both the state labeling and the transition relation. The quotient model induced by this relation is often smaller than the original model. Moreover it is bisimulation equivalent to the original model. Therefore, all queries on the original model can be instead checked on the quotient model.

Partial Order Reduction Partial order reduction 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 [10] [11] [12], in particular, when the interim global states are not relevant to the query being checked, model checkers only need to explore one of the possible execution orders. This may radically reduce model checking complexity.

Asynchronous interleaving semantics of xUML suggest application of Static Partial Order Reduction [13] 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 query to be checked. This enables integrated application of partial order reduction while applying symbolic model checking to the S/R model.

Symbolic Model Checking Symbolic model checking represents the state transition structure of an xUML model with binary decision diagrams, which enables manipulation of entire sets of states and transitions instead of individual states and transitions. This heuristic is fully automatic and has shown significant reduction promise on some xUML models.

2.3 Interactions between Reduction Algorithms

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

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

a. Symmetry reduction is often selected if there exist many instances of the same class;
b. Partial order reduction is often selected if there is intensive execution interleaving;
c. Symbolic model checking is often selected if there is much randomness.
d. Partial order reduction and symbolic model checking are often combined if the conditions in both b and c are true.
e. Localization reduction is always applied on S/R models.

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

Application Order of Reduction Algorithms To maximize the state space reduction effect, it is always attempted to apply each reduction algorithm to the minimum models with which the algorithm has to deal. Therefore, the framework defines some application ordering relations between reduction algorithms:

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

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

2.4 Instantiations of General Framework for Application Domains

System designs from the same application domains commonly follow a set of domain-specific design patterns and require satisfaction of queries in similar formats. Therefore, domain specific design patterns can be explored to establish an instantiation of the general framework for a given domain. The instantiation should provide additional guidelines for selecting reduction algorithms and additional relations for ordering these reduction algorithms in the user-driven reduction procedure. An instantiation of the framework on distributed transaction systems is given in Section 4 as an example.

3 Automation of Integrated State Space Reduction

Automation support, which is crucial to the wide application of the general framework, is provided through selecting an appropriate model checker, extending the xUML-to-S/R translator, and introducing a reduction manager.

3.1 Selection of Model Checker

COSPAN, which has synchronous and parallel semantics, is selected as the model checking engine for xUML models because it supports both symbolic model checking, which is not readily supported by effective model checkers with asynchronous interleaving semantics, and localization reduction. Localization reduction is always applied on any S/R model while symbolic model checking can be switched on or off by setting an option of COSPAN.

3.2 Extension to xUML-to-S/R Translator

The xUML-to-S/R translator inputs an xUML model and an xUML query and outputs an S/R model and an S/R query. Static partial order reduction has been implemented as a preprocessor of the translator that transforms the xUML model with respect to the xUML query before passing the model to the translator. Quotient model reduction is going to be implemented as another preprocessor that will be placed before static partial order reduction. These reduction algorithms can be switched on or off without affecting the translation.

3.3 Reduction Manager

A reduction manager has been designed and is under development, which coordinates the recursive model checking process in Figure 2. If the current subtask is not directly model checkable, the manager invokes a user interface to input:

- Selected reduction algorithms and their application order;
- Sub-queries of a complex xUML query;
- Boundaries and environment assumptions of a system component;
- Correspondence between sub-queries and components (or units of work);
- Class instances involved in a unit of work;
- Abstract state models and their corresponding concrete state models;
- Abstract data types and their mapping relations to concrete data types;
- Symmetric relations between class instances (or queries).

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

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

4 Framework Instantiation on Transaction Systems

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

4.1 Common Patterns of Transaction Systems

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

Definition 1 *The model, M, of a transaction system, S, is the xUML model of S, which consists of a set of interacting class instances. A model, M', is a sub-model of M if M' consists of a subset of class instances of M.*

Definition 2 *A transaction type, T, of M is a message sequence template, which consists of sequences of message types defined in M. An instance of T is a transaction executed by M, whose message sequence follows T. A type, T', is a sub-type of T if each sequence in T' is a sub-sequence of a sequence in T.*

Definition 3 *A transaction property, P, is a temporal logic predicate over all instances of a transaction type, T, or over an instance of T.*

Definition 4 *A model checking task is a tuple, $< M, T, P, A >$, where M is a model, T is a transaction type defined on M, P is a transaction property defined on T, and A is the set of assumed properties of the environment of M. The environment of M is the aggregation of all inputs to M. A model checking task, $< M', T', P', A' >$, is a subtask of $< M, T, P, A >$ if M' is a sub-model of M, T' is a sub-type of T, and P' is a predicate derived from P through reduction, and A' is the union of A and a set of assumed properties of $M - M'$.*

Definition 5 *A model checking task, $< M, T, P, A >$, is directly model checkable if P can be checked on M by a model checker within a reasonable amount of time and memory assuming that all properties in A hold on the environment of M.*

4.2 Domain Specific Reduction Algorithm

The domain specific reduction algorithm for checking a task, $< \hat{M}, \hat{T}, \hat{P}, \hat{A} >$, on a transaction system is given in Figure 3. For simplicity, only the reduction aspect

```
Enqueue(ToDo, < M̂, T̂, P̂, Â >); Done ={ };
Do
  < T, M, P, A > = Dequeue(ToDo);
  If (< T, M, P, A > is Directly Model Checkable) Then
   Model check < T, M, P, A >; Done = Done + {< T, M, P, A >}; Continue;
  End;
  If (P is a query over all instances of T) Then
   Reduce P with Symmetry Reduction to P1 where P1 is on Instance 1 of T;
   Enqueue(ToDo, < T, M, P1, A >); Continue;
  End;
  If (M consists of instances from different classes) Then
   Current = The first class that appears in T;
   Decompose M into M1={All instances of Current} and M2=M − M1;
   Decompose T into T1 performed by M1 and T2 performed by M2;
   Decompose P into P1, ..., Pi on M1 and Pi+1, ..., Pm on M2;
   U1 = {P1, ..., Pi}; U1 = {Pi+1, ..., Pm}; D1 = { }; D2 = { };
   While(!Empty(U1) or !Empty(U2))
     If (!Empty(U1)) THEN
      P' = Remove-an-element(U1); A' = {Assumptions of P' on M2}
      Enqueue(ToDo, < T1, M1, P', A' >); D1 = D1 + {P'}; U2 = U2 + A' − D2;
     End;
     If (!Empty(U2)) THEN
      P'' = Remove-an-element(U2); A'' = {Assumptions of P'' on M1}
      Enqueue(ToDo, < T2, M2, P'', A'' >); D2 = D2 + {P''}; U1 = U1 + A'' − D1;
     End;
   End;
  End;
  If (M consists only of all instances of a class, C) Then
   Reduce M with Case Splitting to M1 where M1 = {Instance 1 of C};
   Enqueue(ToDo, < T, M1, P, A >); Continue;
  End;
Until (Empty(ToDo));
```

Fig. 3. Domain Specific Reduction Algorithm for Transaction Systems

of the algorithm is covered in Figure 3. The algorithm constructs the reduction tree for $<\hat{M}, \hat{T}, \hat{P}, \hat{A}>$ on-the-fly. The root of the tree is $<\hat{M}, \hat{T}, \hat{P}, \hat{A}>$. Each non-root node in the tree is a subtask of its parent. The tree is expanded in a breadth first fashion. Every execution of the do loop either discharges a task at a leaf of the tree or expands the tree by reducing the task into its subtasks through symmetry reduction, decomposition, or case splitting. The expansion stops when all subtasks at the leaves of the tree are directly model checkable.

4.3 Case Study: An Online Ticket Sale System

The xUML model of an online ticket sale system [14], M_0, is employed to illustrate the domain specific reduction algorithm for transaction systems. There are four classes in the system: Customer, Dispatcher, Agent, and Ticket Server. Both the Dispatcher class and the Ticket Server class have only one instance. The Agent class and the Customer class may have an arbitrary number of instances.

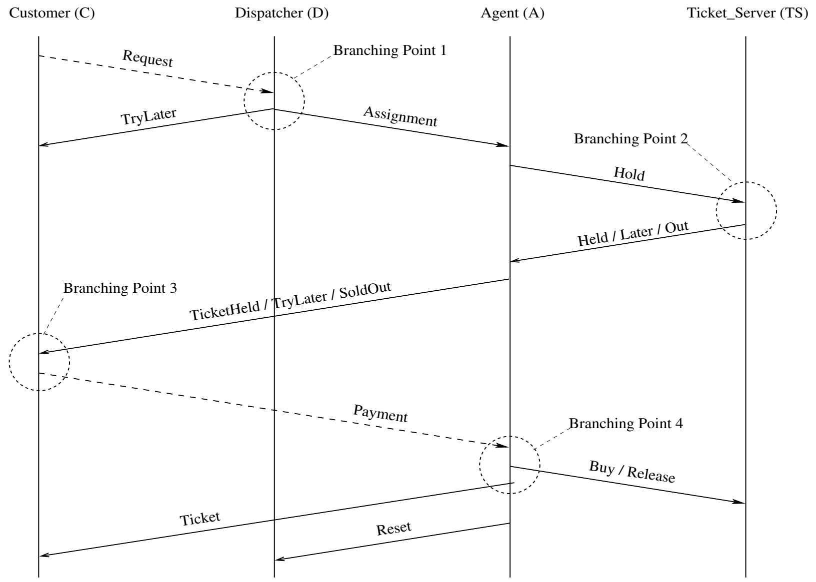

Fig. 4. Message Sequence Diagram of Ticketing Transaction

The system processes ticketing transactions of the type, T_0, concurrently for many customers. The message sequence diagram of T_0 is shown in Figure 4. T_0 has four branching points where the decisions made affect the message sequences:

1. Upon processing a *request* message from a customer, the dispatcher assigns an idle agent to the customer if there is an idle agent; Otherwise, replies to the customer with a *TryLater* message.
2. Upon processing a *Hold* message from an agent, the ticket server replies to the agent with: A *Held* message if the number of tickets available is greater than the requested number; A *Later* message if the sum of tickets available or being held is greater than the requested number; An *Out* message otherwise;
3. Upon receiving a *TicketHeld* message from an agent, the customer may or may not reply to the agent with its payment;
4. If the valid payment from the customer is received before the agent times out, the agent sends a *Ticket* message to the customer and a *Buy* message to the ticket server; Otherwise, it sends a *Release* message to the ticket server.

A property which should hold on each transaction of the type, T_0, is that after a *request* message from a customer is processed by the dispatcher, eventually there will be a *TicketHeld* message, or a *TryLater* message, or a *SoldOut* message sent back to the customer. The property is formulated as P_0 in Figure 5.

Although the structure of the system is simple, the arbitrary number of customers and agents make directly model checking P_0 infeasible even for the most powerful model checkers. Therefore, the domain specific reduction procedure is applied to reduce the model checking task, $< M_0, T_0, P_0, \Phi >$. The assumption set is empty since customers are also modeled as class instances in M_0. The sub-queries involved in the reduction process are defined in Figure 5. The sub transactions and the sub-models involved in the process are shown in Figure 6. The reduction tree generated by the process is shown in Figure 7. Assumptions

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

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

P_{21}: **After** Request(1) and **Forall** i, D.Agent_Free[i] = FALSE
Eventually TryLater(1)
P_{22}: **After** Request(1) and **Exists** i, D.Agent_Free[i] = TRUE
Eventually Assignment(j, 1) and A(j).$ = Idle
P_{23}: **After** Assignment(j, 1) and A(j).$ = Idle
Eventually TicketHeld(1) or TryLater(1) or SoldOut(1)

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

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

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

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

Fig. 5. Original Query and All Intermediate Sub-queries

of a subtask are represented in Figure 7 by dashed arrows which lead to the subtasks that check the assumed properties on the corresponding sub-models. Reductions applied in the process are grouped into six general steps as follows:

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

Step 2: Decomposition T_0 is decomposed into three sub transaction types, T_{11}, T_{12} and T_{13}. Accordingly, M_0 is decomposed into three sub-models, M_{11}, M_{12}, and M_{13}. Transactions of the types, T_{11}, T_{12} or T_{13}, are conducted by M_{11}, M_{12}, or M_{13} respectively. P_1 is decomposed into three sub-queries: P_{21}, P_{22}, and

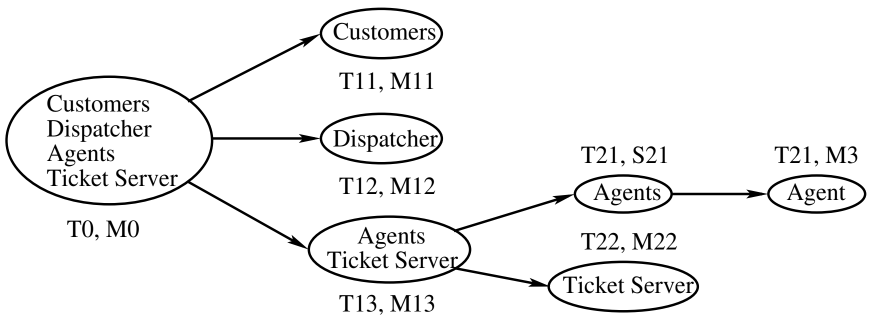

Fig. 6. Decomposition Relations between Sub-models Involved in Reduction

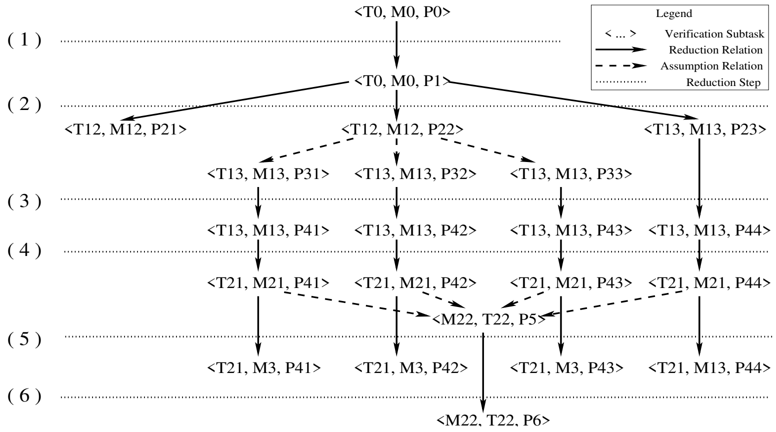

Fig. 7. Reduction Tree for Verifying P_0 on Online Ticket Sale System

P_{23}. P_{21} is directly model checkable on M_{12} without any assumption on M_{11} or M_{13}. P_{22} is directly model checkable on M_{12} by assuming that P_{31}, P_{32}, and P_{33} hold on M_{13}.

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

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

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

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

5 Evaluation of Integrated State Space Reduction

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

5.1 Evaluation of User-Driven Reduction Algorithms

Statistics from model checking Property P_0 in Figure 5 on the xUML model of the online ticket sale system are employed to demonstrate the effectiveness of the user-driven reduction algorithms. The memory and time usage for directly model checking P_0 is compared with the memory and time usage for checking the subtasks generated by applying these reduction algorithms.

Directly model checking P_0 on the xUML model with two customer instances and two agent instances requires two separate model checking runs for $i = 0$ and $i = 1$. With state partial order reduction (SPOR) and symbolic model checking (SMC) applied, each run takes 152.79 megabytes and 16273.7 seconds. The complexity of the xUML model increases rapidly as the number of customers increases. Directly model checking P_0 on the xUML model with 6 customer instances cannot be fulfilled. Therefore, directly model checking the xUML model with arbitrary number of customers is not feasible.

The memory and time usage for model checking each subtask from the reduction tree in Figure 7 is shown in Table 1. It can be observed that the memory

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

Table 1. Time and Memory Usage of Subtasks in Verifying P_0

and time usage for each subtask is substantially lower than that for directly model checking P_0 on the xUML model with two customers. More importantly, the model checking result from the reduction process can be scaled up to xUML models with arbitrary number of customer and agent instances.

The complexity of symmetric query reduction is not shown due to an unfinished feature of our reduction system. However, the complexity is theoretically bounded by the complexity of the static structure of an xUML model because the reduction only checks the static structure of the model instead of exploring the full state space of the model.

5.2 Evaluation of SPOR, SMC, and Their Combined Application

Being able to directly discharge larger model checking tasks reduces user interventions and makes the integrated state space reduction more automatic. Currently, to scale up directly model checkable tasks, SPOR is applied in the xUML-to-S/R translation and SMC is applied in the S/R level model checking. To demonstrate the reduction ability of SPOR and SMC, Property P_{21} in

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

Table 2. Model Checking Memory and Time Usage Comparison

Figure 5 is directly checked on the whole model under the four possible on/off combinations of SPOR and SMC. The model checking complexities under the four combinations are compared in Table 2. It can be observed that both SPOR and SMC lead to significant reduction on the model checking complexity. SPOR offers a better memory usage while SMC offers a better time usage. The combined application of SPOR and SMC achieves the best time usage with a medium memory usage.

6 Conclusion

This paper defines and describes a general framework for integrated state space reduction in model checking executable object-oriented software system designs. The framework is presented for system designs modeled in xUML, but is readily applicable to other representations. Partially implemented automaton support for the framework is discussed. The framework is illustrated by its instantiation for distributed transaction systems and is evaluated by applying the instantiation in model checking an online ticket sale system. The dimension of the software system designs that are model checkable is found to be greatly extended.

7 Acknowledgement

We gratefully acknowledge Robert P. Kurshan, Vladimir Levin, Huaiyu Liu, Nancy Macmahon, and Kedar Namjoshi for their generous help.

References

1. Kennedy Carter: http://www.kc.com/html/xuml.html. Kennedy Carter (2001)
2. Project Tech.: http://www.projtech.com/pubs/xuml.html. Project Tech. (2001)
3. Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. Logic of Programs Workshop (1981)
4. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. The MIT Press (1999)
5. Xie, F., Levin, V., Browne, J.C.: Model Checking for an Executable Subset of UML. Proc. of 16th IEEE International Conf. on Automated Software Engineering (2001)
6. Hardin, R.H., Har'El, Z., Kurshan, R.P.: COSPAN. Proc. of 8th International Conf. on Computer Aided Verification (1996)
7. Cadence: FormalCheck User Guide. Cadence (2001)
8. McMillan, K.L.: A Methodology for Hardware Verification Using Compositional Model Checking. Cadence Technical Report (1999)
9. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press (1994)
10. Godefroid, P., Pirottin, D.: Refining Dependencies Improves Partial-Order Verification Methods. 5th International Conf. on Computer Aided Verification (1993)
11. Peled, D.: Combining Partial Order Reductions with On-the-fly Model-Checking. Formal Methods in System Design (1996)
12. Valmari, A.: A Stubborn Attack on State Explosion. Proc. of 2th International Conf. on Computer Aided Verification (1990)
13. Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigün, H.: Static Partial Order Reduction. Proc. of 4th International Conf. on Tools and Algorithms for the Construction and Analysis of Systems (1998)
14. Wang, W., Hidvegi, Z., Bailey, A.D., Whinston, A.B.: E-Processes Design and Assurance Using Model Checking. IEEE Computer Vol. 33 (2000)