Statecharts Reduction and Composition with Properties

Zhiwei An Dennis Peters

Faculty of Engineering and Applied Science
Memorial University of Newfoundland
St. John's NL A1B 3X5

zhiwei@engr.mun.ca *dpeters@engr.mun.ca*

November 08, 2005

Abstract

In software design, UML statechart and OCL are used to describe component behaviors and the interactions between components. To verify the design, model checking technology is used and properties are extracted from the software requirements. When composing statecharts directly, the state space explosion problem cannot be avoided. In this paper, we will discuss how to use properties from the software requirements to reduce the statecharts and how the reduction makes it possible to compose statecharts for model checking. To check all properties, different properties are used one by one in the reduction and composition so the verification models contain a set of reduced and composed models to be checked.

1 Introduction

In software design, the UML/OCL model is often used to describe the component behaviors and the interactions between components. OCL has the power of describing variables and invariants in the component, and it can also specify pre- and post-conditions of the access programs. Moreover, OCL expression also describe the states in one component. From the standard of a module interface specification, OCL could be used as a module interface specification language to describe the state variables, the signatures of the functions, the behavior of the functions and the output values. UML statechart has been successfully used in dynamic modeling and it can be used together with OCL to describe more complex behaviors of components and connectors.

In the design with UML/OCL, the main problem is the correctness of the design documents. Verification methods including model checking are adopted to find and remove the errors in the design. The main idea of model checking is doing state space exploration and checking if some properties are held.

In model checking, it takes a long time to check all the states and it is impossible to perform the algorithm in a reasonable time when the number of states is too large. In component based systems, the state space explosion problem is more protrude because the number of states in the whole system is the product of the number of states in all components.

There are several methods used to deal with the state space explosion and the examples are symbolic representation of states and states transitions [1], state refinement [2], state abstraction [3] and slicing technology [4]. These methods have each been successfully used to resolve some particular applications.

In this paper, we propose a method of verification based on the properties derived from software requirements written in a formal language. Properties are normally described by the the changes of certain variables so checking this property is actually checking the changes of the variable(s). It is possible to only consider related variable(s) during one check so we can consider the related variable(s) when transforming and then composing the reduced statecharts. In Section 2, A brief discussion of this method will be presented. The input design model will be discussed in Section 3. In Section 4, we will discuss how to derive properties are

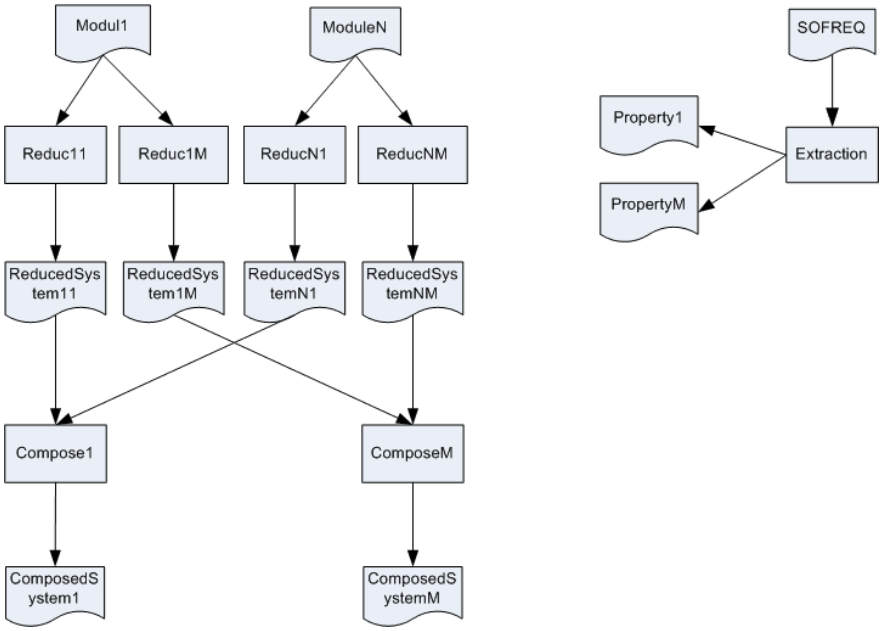

Figure 1: Technology Profile

derived and how we can derive them from software requirement. The main idea and steps of reduction and composition will be discussed in Section 5.

2 Technology Profile

In model checking, design is checked whether it holds some given properties. Properties are the constraints a system must follow. Examples of properties are *free of deadlock*, *livelock* or some other more complex ones in logic expressions. If the properties are not hold in the design, it is said that the design is not compliant with the requirements.

Properties are normally from software requirement. Software requirement can be written in a formal language such as [5]. Logic expression is a formalism and it can be derived from software requirement. Some attempts to do this have been discussed in [6].

When describing one property, either the required states' changes or some required sequences of events are considered. The sequence of the events is equivalent to the changes of the states because the events cause the changes of the states.

States of the system are determined by the values of all state variables defined in the design documents. One state can be written as $(var_1, var_2, \ldots var_n)$ where n is the total number of state variables in the design. Normally, the number of variables related to one property is much fewer than n. This phenomenon makes it possible for us to only consider the related variables and transform the statecharts according to these variables.

In component based systems, properties could be both global and local. Properties describe the required states' changes of the whole system so they are global to the whole system. At the same time, one state variable only belongs to one component so the check of one property may be limited to a small number of components.

In Figure 1, properties *1* to m are derived from SOFREQ. In the system with n components and connectors, one property is adopted to transform all components. Then the reduced components and connectors will be composed together for further model checking. With another property, the different transformation can be performed to generate different reduced statecharts for composition.

3 UML/OCL Design Model

3.1 OCL as a Module Interface Specification Language

OCL is a formal language filling the gap between UML specification and full formal method. OCL is usually used to describe variables, invariants, guards, etc..

A typical module interface language consists of the signatures of functions, descriptions of state variables, state transition functions and output functions. OCL can be used as a module interface specification language from the following aspects.

- The invariant of in a component could be defined with the keyword *inv*.

  Context Server
  inv numToRead:int, numToRead$\leq$ numTotal AND numToRead ≥ 0

- The state variable of a component could be defined with the keyword *def*.

  Context Server
  def:servedReaders : Set(String)

- The signature of a function could be like

  Context Server::read(String: readerID): String
  pre : readerID $\notin$ servedReaders
  post: result = messageInBuffer

  In this context, output values are also described by the post-condition of the function with the keywords *post* and *result*. The keyword *post* can also decribe the state transition after the functions is called.

3.2 Statechart for Behavioral Modeling

Statechart in UML is one kind of diagram to describe dynamic aspect of a component or connectors. A statechart diagram consists of states and transitions. The states in statechart represents the current status of the components and the transitions describes the event, guard of the events and effect of the event. In states, the entry, exit event are defined to describe what happens when the system enters or leaves this state. Composite state is also used to describe the hierarchy modeling. Statechart has the same modeling power with a more concise notations. In statechart diagram, OCL function *oclInState* could be added to represent the relationship between a component state and the values of state variables. For example, $inv : self.oclInState(Writing)\ implies\ numToRead = 0,\ servedReaders = \{\}$

Statechart and OCL compensate the weakness of each other and combined as a combinational method in component specifications and interaction description.

4 Extracting Properties from Software Requirements

The properties are some constraints that a system must satisfy and are normally expressed in a notation such as combinational logic or temporal logic. Combinational logic uses $\forall, \exists, \rightarrow, \wedge$ *and* $\vee$ to describe the boolean values of formulas at one time point. Temporal logic is divided into linear time and branching time logic. In linear time logic, operators *X, U, F, G* are used to describe *next, until, eventually and always.* In branching time logic, *A* and *E* are added to show all branches or some branches.

The properties of interest are found in the software requirements document, which, as discussed in [5], describes the relation SOFREQ. These requirements could be written in one of several different formalisms

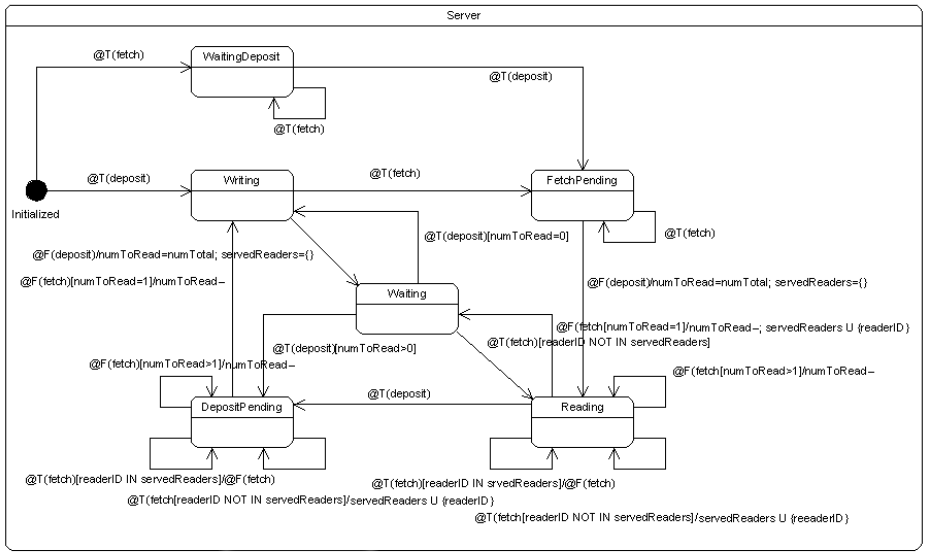

Figure 2: Component Server

such as [5]. In this work we use a variation of the Software Cost Reduction (SCR) method [5], which is a requirements specification method describing interface, behavioral, functional, precision, and timing requirements. The SCR method offers a tabular notation for specifying requirements. Underlying this notation is a state machine model. In SCR, environmental conditions, mode classes, and modes are defined with variables. The constraints on each of them and transitions between them are logic expressions.

The procedure of extracting properties from software requirements is a preliminary step of the reduction and composition. The SCR method and logic expressions have the same power for expressing the properties so the extraction is feasible with a transformation. The extraction step could be considered in two categories, as follows:

- Combinational logic expressions can be used for those constraints that are related to values in one or more particular states, but are not related to state transitions. Constraints on mode, mode class and enviromental conditions often fall into this category.
- Logic expressions using temporal quantifiers such as *always (G)*, *until (U)*, *eventually (F)* and *next (X)* can be used for those constraints related to mode or state transitions or sequences of events.

5 Reduction and Composition

The reduction is based on the variables from properties. The first step of reduction is establishing the relationship between properties and variables. After this step, we can know what vairble(s) we can choose when checking one property. The second step is selecting one property, splitting and merging all states according to the variable(s). The third step is to determine the irrelevant events to the property and generate the new state machine. The example component is in Figure 2.

- Search all states in the statechart and split all states in statechart.
  - *Reading* could be divided into *numToRead=1* and *numToRead=2*. This is from the OCL expression
    self.oclInState(Reading) implies 0<numToRead<=numTotal; servedReaders $\in$ $Readers^*$.

- *Waiting* could be *numToRead=2*, *numToRead=1* and *numToRead=0*.
  self.oclInState(Waiting) implies 0<=numToRead<=numTotal; servedReaders ∈ $Readers^*$.
- *DepositPending* could be divided into *numToRead=2* and *numToRead=1*
  self.oclInState(Reading) implies 0<numToRead<=numTotal; servedReaders ∈ $Readers^*$.
- *Initialized*, *WaitingDeposit*, *Writing* and *FetchPending* are the same as *numToRead=0*

- Merge the states.

  In this example, the states are merged according to the value of *numToRead*.

- Ignore irrelevant events

  - If one event changes the value of the variable, it must be a directly relevant event.
  - If one event must happen before another directly event can happen but it does not change the value of the variable, it is also a relevant event.
  - If an event does not change the value of the variable and it is not a prerequisite event for the above two kinds of events. It is a totally irrelevant event.

With the reduction method above, the components without the specified variables will become the stateless components. This does not mean that they do not affect behavior of the system. On the contrary, these components add constraints to those components with states. Although no state is in such components, the events happen in these components can still happen and events affect the components with states by firing the events of the access programs in component with state.

6 Conclusion and Future Work

From the above discussion, we can conclude that the statecharts of the components can be transformed and reduced to smaller ones with much fewer number of states and some components can even be viewed as stateless components. With this technology, the model checking algorithm can deal with larger systems with more states.

Future steps of this project include considering more types of communications and considering symbolic technology in the reduction. Also, whether the algorithm for reduction holds the property is being approved.

References

[1] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic model checking for real-time systems," in *Proc. IEEE Symp. Logic in Computer Science*, pp. 394–406, IEEE Computer Society Press, 1992.

[2] J. Woodcock and J. Davies, *Using Z : Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[3] R. Bharadwaj and C. L. Heitmeyer, "Model checking complete requirements specifications using abstraction," Tech. Rep. NRL/MR/5540–97-7999, Naval Research Laboratory, Center for High Assurance Systems, Washington, DC 20375-5320, Nov. 1997.

[4] M. Weiser, "Programming slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 352–357, July 1984.

[5] D. K. Peters, *Deriving Real-Time Monitors from System Requirements Documentation*. PhD thesis, McMaster University, Hamilton ON, Jan. 2000.

[6] J. M. Atlee and M. A. Buckley, "Logic-model semantics for SCR software requirements," in *Proc. Int'l Symp. Software Testing and Analysis (ISSTA '96)*, pp. 280–292, ACM SIGSOFT Software Engineering Notes, vol. 21, no. 3, May 1996.