Chapter 2 Background

With the inevitable increase in complexity of both hardware and software systems, the likelihood of subtle errors is high. In the industry, many demand for methodologies or tools that can increase confidence in the correctness of system design and construction. Such methodologies or tools could produce better quality software, as well as in a reduction to the total development cost of a system. In this section, we describe the background of software verification and related researches.

2.1 Model checking

One traditional engineering approach to the construction of complex systems is to build models before real implementation. Models can be studied and modified until their correctness can be assured to some extent. The advantage is that models are simpler, and their development cost is negligible when compared to the cost of building the system itself. In model checking, a finite model of a system is built and checked against a set of desired properties. Model checking is fast and fully automated. The system model is in essence a finite-state machine, which is intuitive to the average engineer. The system may be expressed directly in terms of state machines. Alternatively, a subset of some higher-level language may be used, which

permits more concise specifications, while restricting the developer to finit-state models that can be handled by the model-checking approach.

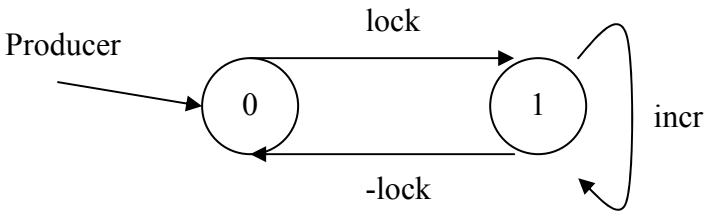
In model checking, desired properties are usually expressed either in some temporal logic [21] or in terms of automata [22]. An exhaustive search of the state space is performed in order to check that the system is a model of its specifications. This search is guaranteed to terminate, since the model is finite. When both the system and its specifications are modeled as finite-state machines, the system can also be compared to the specification to determine whether its behavior conforms to that of the specification.

However, modeling complex systems as finite-state machines has an inherent disadvantage, commonly known as *state explosion*. The problem describes the exponential relation of the number of states in the model of a system, to the number of components of which the state is made. Model checking cannot handle efficiently systems that are made up of a large number of state machines. A number of methods have been proposed to solve state explosion problems, and compositional analysis is the better one. We will describe in latter section.

2.2 Communication finite state machine (CFSM)

For automatic verification, a *communicating finite state machine* (CFSM) is used to model the behaviors of a synchronous communicating process in a concurrent

system. A CFSM contains all the states the process may reach and all the transitions it may perform. Usually, we refer to CFSM as a state graph. Take *Producer* for example,



state 0 is an initial state which contains a transition without any label. *Producer* can translate its current state from state 0 to state 1 through transition "lock" which represents a behavior of Producer. Formally, a CFSM of a process P is a quadruple (*S, A,* Δ*, s_0*) where

- *S* is a set of state
- $A = \alpha P \cup \{\tau\}$, where αP is the communicating alphabet of *P* which does not contain the internal action τ
- $\Delta \subseteq S \times A \times S$, denotes a transition relation
- s_0 is a state in *S* which indicates the initial state of *P*.

An CFSM of *P* = (*S, A,* Δ*, q*) transits into another CFSM of P' = (*S, A,* Δ*, q'*) with an action a in αP if and only if (*q, a, q'*) ∈ Δ and *q*, *q'* ∈ *S*. That is,

$$(S, A, \Delta, q) \xrightarrow{a} (S, A, \Delta, q') iff (q, a, q') \in \Delta$$

We use $P \xrightarrow{a} P'$ to denote the translation between states.

2.3 Promela and rc-Promela

Because COCOV use rc-Promela as the front-end language of refactoring, and

rc-Promela is a subset of Promela, where “r” stands for “refactoring” and “c” stands for “ccs”, we need to describe the semantics of Promela. Promela (Process or Protocol Meta Language) is a modeling language which used to describe the behaviors of a system. This language allows for dynamic creation of concurrent processes. Communication via messages channels can be defined to be synchronous (i.e., rendezvous), or asynchronous (i.e., buffered). Promela programs consist of processes, message channels, and variables. Processes in Promela are global objects, message channels and variable can be declared either globally or locally within a process. The syntax of Promela is C-like. We give an example of Promela codes in Figure 2.1.

In line 1-2 are data structure definition, such as channel or global variables declaration. The keyword *mtype* definition is used to declare symbolic constants. Only one *mtype* definition is allowed in Promela codes which must be global and at most 255 symbolic constants can be declared. In line 2, the keyword *chan* is used to define message passing between processes. The communicating way of the channel named *sema* is synchronous; it indicates sender and receiver must synchronize as no message can be stored. If the buffer of the channel greater than 0, the communicating way between processes is asynchronous. In line 3-11 is process declaration. Use the keyword *do* to show loop declaration, the keyword *od* represents the end of this control block. In line 7 is channel output statement, it means variable *p* sends a message via channel *sema*. In line 9 is channel input statement, it means variable *v* receives a message via channel *sema*. In line 20, the keyword *init* is used to initialize variables and running the appropriate process-instances. Because this communicating

way is synchronous, thereby, when variable *p* sends a message, variable *v* is received right away to do synchronization.

```
1 mtype = {p,v}
2 chan sema = [0] of { mtype};
3 proctype dijkstra(){
4         byte count = 1;
5         do
6         :: (count == 1) ->
7                   sema!p;    count = 0;
8         :: (count == 0) ->
9                   sema?v;    count = 1;
10        od
11 }
12 proctype user(){
13        do
14        :: sema?p;
15           /*         critical section */
16           sema!v;
17           /* non-critical section */
18        od
19        }
20        init{
21                  run dijkstra();
22                  run user();
23                  run user();
24        }
```

Figure 2.1 Codes for Promela

This example describes a simple program that using Promela to simulate semaphore. This example has two user and one dijkstra process. Process dijkstra use variable *count* to control the process. Initially, the value of variable *count* is 1, therefore, process *dijkstra* outputs a message and flip the value of *count*. The two processes *users* both have the identical behavior are competed to receive this message. Only one *user* can receive the message and sends a message for process *diskstra*.

When *dijkstra* receives this message, flips the value of *count* and go back to the loop again. Thus, the two *user* processes compete to get the message repeatedly.

2.4 Model checking tools

A large number of model-checking tools have been developed over the years. This section provides an overview of some well-known model checkers.

SMV

SMV [23] is a tool for checking finite-state systems against specification in the temporal logic CTL. It supports a flexible specification language and use an OBDD-based symbolic model-checking algorithm for efficiently checking whether CTL specifications are satisfied by the system. The tool has been used to verify several industrial designs such as the Futurebus+ and the Gigamax protocol.

SPIN

SPIN [13] is a state-based model-checking tool designed for the verification of distributed systems. Its native specification language is Promela, whereas its semantic model is based on finite automata. Correctness requirements can be expressed directly in LTL (Linear-Time Temporal Logic)

LTL formulas are automatically translated into Promela “never-claims”, which represent the Büchi automaton corresponding to the negation of these formulas. SPIN performs model checking “on-the-fly”. To this end, it uses an efficient depth-first

search algorithm that is compatible with all modes of verification supported by the tool, i.e. exhaustive search, bit-state hashing and partial-order reduction techniques. These techniques, together with state compression are used for dealing with large state spaces.

2.5 Related software verification tools

Bandera

Bandera [8] is a tool for model checking Java source code using a component-based architecture. It uses slicing to automate irrelevant component elimination. Slicing is that given a program *P* and some statement of $C = \{s_1,...,s_k\}$ from *P*, we reduce *P* by removing statements of *P* that do not affect the criterion statement *C*. Bandera can construct models of the source code written in Java, then using model checking tools such as SPIN to check the models if on the property.

JavaPathFinder

Since Promela is a very rich model description language the semantic gap is not nearly as great as programming languages. JavaPathFinder is a translator from a subset of the Java programming language to Promela. A Java program may contain assertions, which are translated to similar assertions in the Promela model. Others like dynamic creation of objects with data and method, class inheritance, threads, exceptions, etc, JavaPathFinder is also supported. The goal of JavaPathFinder is to

automatically verify program written in real programming language such as Java.

2.6 Compositional analysis

In a compositional analysis, we often have to group a set of processes into a subsystem (or a model). There are two basic criteria of a "good" subsystem. First, the processes inside the subsystem must not generate excessive state space. Second, the subsystem's state space must be able to be replaced by a much simpler *interface process* to represent the subsystem's state space. An interface process can be computed automatically by hiding internal interactions, minimizing the state space, and exporting the state and transitions (a.k.a interfaces) that will be used by its environment. Note that exporting state and transitions as interface can aggregate the state explosion problem if the interfaces are not simple (see [9]). So, simple interface is the key to a "good" subsystem. In other words, an effective subsystem should be *loosely coupled* to its environment so that the chance of having a simple interface process to replace it in compositional analysis is higher. At last, "good" subsystems and processes must produce another larger "good" subsystem in the compositional hierarchy until the whole system is analyzed. Unfortunately, this ideal scenario seldom occurs in the compositional analysis of lager and complicated systems.

In Figure 2.2 and Figure 2.3, we show the state graph of three example processes *X*, *Y*, and *S* in CCS semantics[17] (where synchronization actions are matched in

pairs) and their synchronization structure. Such kind of structure, a star-shape structure, appears often in practice, for example, a stateful server which communicates with clients via separate (or private) channels. Many systems can even have structures of multiple stars.

In Figure 2.2, *S* is *tightly couple* to its environment (consisting of *X* and *Y*) because it has complicated interface to its environment. Suppose *S* is a server and *X*, *Y* are clients. Image the number of clients is increased to a large number. Any attempt to include S as a subsystem is bound to fail because of the complicated interface to its environment. That is, no feasible subsystems and composing hierarchy exist in this structure, particularly when client number is large.

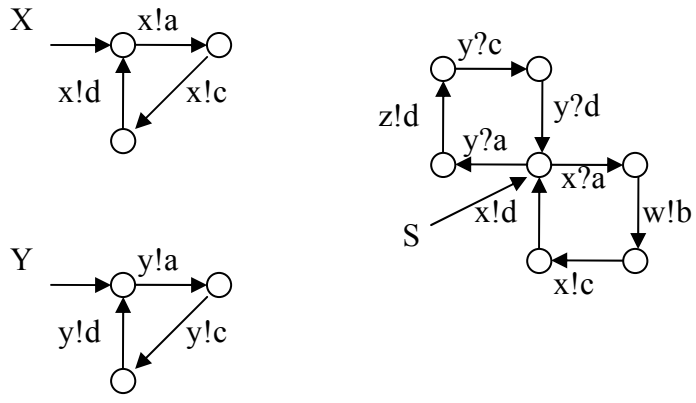


Figure 2.2 A simple example with 3 processes X, Y, and S.

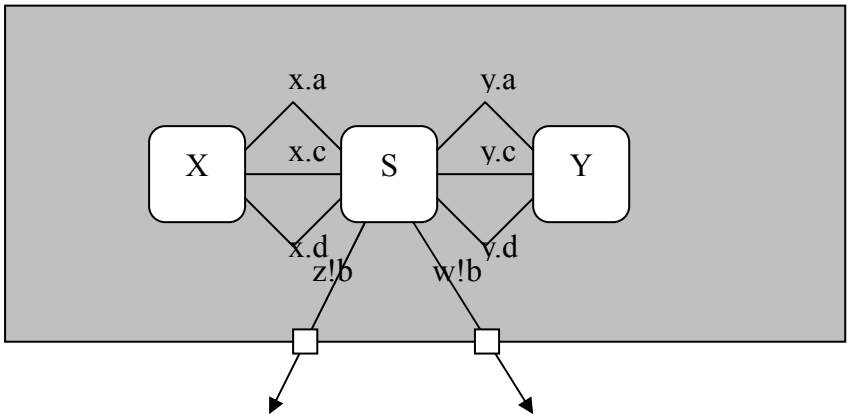


Figure 2.3 The structure of the example.

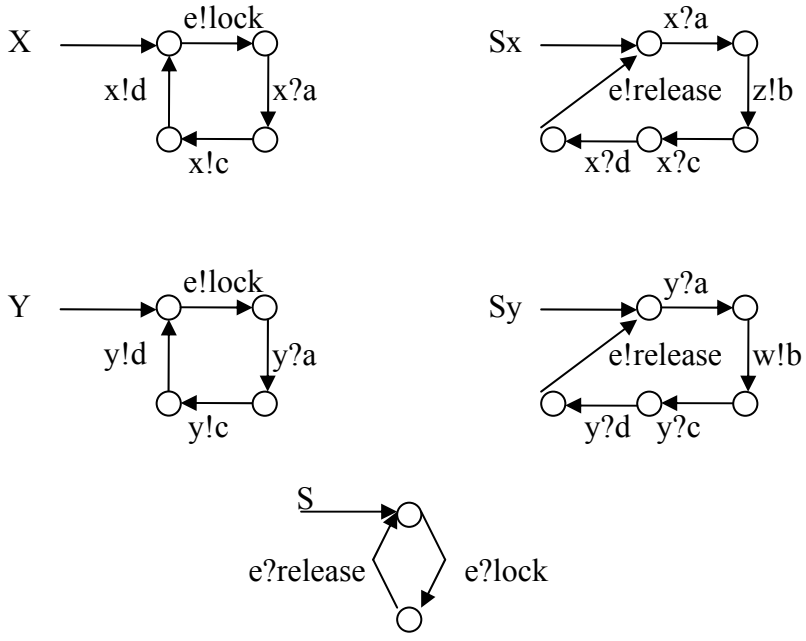


Figure 2.4 The refactored example system.

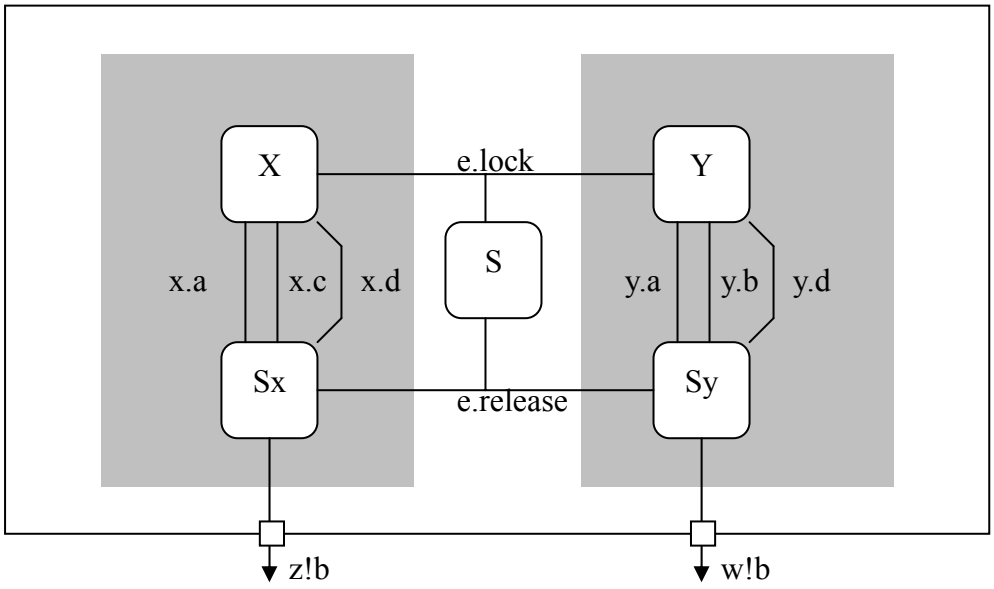


Figure 2.5 The refactored example system.

2.7 Model Refactoring

Recall that the purpose of refactoring is for obtaining, starting from a mode *P*, a semantically equivalent one, which is "more amenable to compositional analysis" than *P*. It consists in building a sequence of equivalent models, each obtained by the preceding ones by means of the application of a rule. The rules are aimed for restructuring the as-built structures which are not suitable for compositional techniques. The goal is to obtain a transformed model whole structure contains loosely coupled component, where processes in each component do not yield state explosion.

For instance, we show the refactored *X*, *Y*, and *S* in Figure 2.3 and the new

synchronization structure in Figure 2.5. In Figure 2.4, the behaviors related to channel *x* (or process *X*) is removed and wrapped into a new process *Sx*. Similarly, the behaviors related to channel *y* is removed and wrapped into a new process *Sy*. So, the rendezvous of *x!a*, *x!c*, and *x!d* are now redirected to *Sx*. However, *Sx* and *Sy* are now two individual processes which can execute concurrently, but their original joint behavior in *S* can not. So, extra synchronizations (*e!lock* and *e!release*) are inserted to maintain behavioral equivalence; that is, before invoking *x!a* and *y!a*, *X* and *Y* are forced to invoke *e!lock* first. Then, at the end of *Sx* and *Sy*, *e!release* is used to free *S*.

The idea of refactoring equivalence is easy to explain. Let's image the modified processes (*X*, *Y*, and *S*) are contained in a black box. Image you are an external observer of the black box. The external behavior of the black box are defined by *z!b* and *w!b*. In Figure 2.3, the black box (which we call it B1) is implemented by 3 processes. The block box (we call it B2) in Figure 2.5, on the other hand, is implemented by 5 processes. The external behaviors are also defined by *x!b* and *y!b*. Our refactoring must ensure the external behaviors are equivalent before and after a transformation. Intuitively, B1's external behavior can be viewed as a specification. Then, we choose to implement the specification with 5 processes. Since we use 5 processes to do the same work which was originally done by 3 processes, extra communications for process coordination are inevitable. As long as the extra

synchronizations are restricted inside the block box, the two block boxes behave equivalently to an external observer.