Chapter 2. Background

2.1 Labeled Transition System

For automatic verification, a communicating finite state machine (CFSM) is often used to model the behaviors of concurrent systems to represent the communicating processes. We usually refer to CFSM a state graph in which contains all states a process may reach and all transitions it may perform. In this thesis, we use *Labeled Transition System* (LTS for short) of a kind of well-known CFSM to represent the behavior of a model.

A formal definition of LTS of a process P is given as a quadruple $<S, A, \rightarrow, q>$, where

- S is a finite set of states;
- $A=A' \cup \{\tau\}$, where A' is the communicating action alphabet of P witch does not contain the internal action τ;
- $\rightarrow \subseteq S \times A \times S$, is a mapping from a state and an action onto another state. $s \xrightarrow{a} s'$ stands for $(s, a, s') \in \rightarrow$;
- $q \in S$ is the initial state of P.

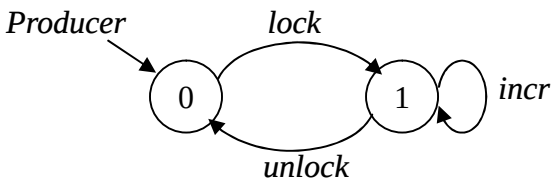


Figure 2.1. An LTS of *Producer* process.

Take *Producer* in Figure 2.1 for example. *Producer* has 2 states labeled 0 and 1 respectively, and state 0 is the initial state. There are three transitions including $0 \xrightarrow{lock} 1$, $1 \xrightarrow{incr} 1$, and $1 \xrightarrow{unlock} 0$. The action alphabet contains *lock*, *incr*, and *unlock*. The behavior of *Producer* starts from state 0 which can perform *lock* action to reach state 1. Then, at state 1, it can execute *incr* action successively several times, or do *unlock* action to go back to state 0.

2.2 Parallel Composition

The difference between CFSM and general FSM is that a CFSM usually has no accepting state, and the actions labeled on transitions represent communications or interactions with other CFSMs. The communicating action of a CFSM can be matched with same actions of other CFSMs, and is served as a communicating bridge among CFSMs. All communicating behaviors of a concurrent system can be obtained using parallel composition of CFSMs, the so-called process of *enumerating the state space*. Let " $\|$ " be the parallel composition operator. Below we present two well-known communicating manners, multi-way and two-way rendezvous, on CFSMs.

2.2.1 Multi-way Rendezvous

Actions communicated with multi-way rendezvous are simultaneously performed while all CFSMs are ready to do the same action. In other words, a synchronous action a may be hold on many CFSMs. In order to trigger the action a in parallel composition, one must wait for other CFSMs all reaching the state which can perform action a. *Communicating Sequential Process* (CSP semantic for short) [5] is the famous representative of multi-way rendezvous. We present an example in Figure 2.2 to compose *P1*, *Q1*, and *R1* in CSP semantic. A formal definition is given as Figure 2.3 which means that if (P_0, a, P_1) is a transition of P and (Q_0, a, Q_1) is a transition of Q, then there must be a transition of (P_0Q_0, a, P_1Q_1) in $P \parallel Q$.

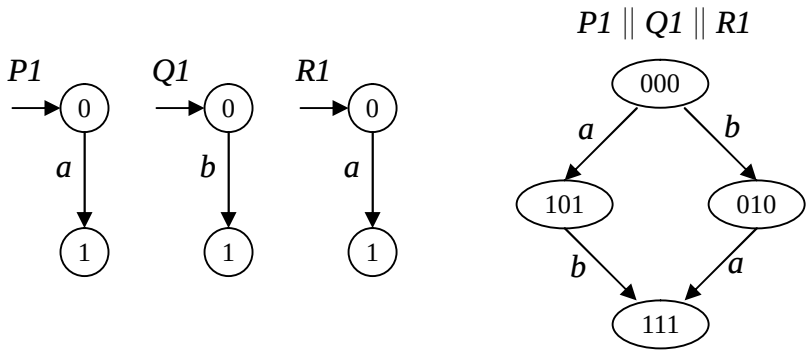


Figure 2.2. Parallel composition of *P1*, *Q1*, and *R1* in CSP semantic.

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P' \parallel Q'}$$

Figure 2.3. Multi-way rendezvous semantic.

2.2.2 Two-way Rendezvous

In contrary to multi-way rendezvous, same *client* actions in two-way rendezvous compete to make a pair with the *server* action. Take Figure 2.4 for example. The server action is denoted as *-a*, while the client actions are denoted as *a*. Client actions, may be more than one, have to compete to invoke the server action. And the matching pair will become an internal action denoted as *τ*. *Calculus of Communicating System* (CCS semantic for short) [4] is a typical notion of two-way rendezvous, and is often useful to represent the behavior of software. A formal definition is given as Figure 2.5.

The internal behavior generated from composing in *CCS* semantic is useful for minimizing a CFSM. The internal behaviors are those unnecessary actions for external communication with other processes. Therefore, we can take away most of them by minimization, but preserve equivalent relations to the original model.

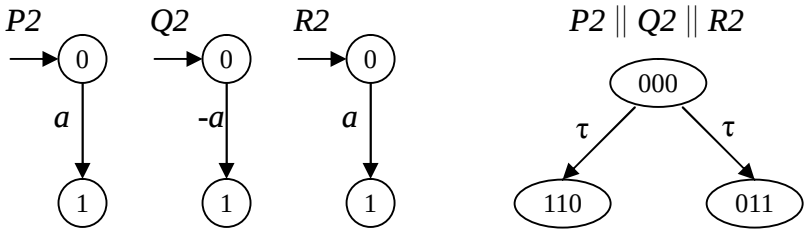


Figure 2.4. Parallel composition of *P2*, *Q2*, and *R2* in CCS semantic.

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{-a} Q'}{P \parallel Q \xrightarrow{t} P' \parallel Q'}$$

Figure 2.5. Two-way rendezvous semantic.

2.3 Behavioral Equivalence

Behavioral Equivalence is an important notion to distinguish whether the *observable behaviors* of two processes are equivalent according to some kind of mathematical definition. The behavior of a process represents a trace of actions it can perform. Take *Producer* in Figure 2.1 for example. One of the possible traces of its actions may be {*lock, incr, incr, incr, unlock, lock, incr, unlock,* …}. We say that two processes are behavioral equivalent, if each of them can perform whatever traces of actions the other does. If we can find another LTS *Q* which performs whatever actions *Producer* does, then we can say that *Q* is behavioral equivalent to *Producer*.

According to different purposes, we apply different definitions of behavioral equivalence. In order to minimize a process to obtain a smaller but behavioral equivalent model, often we take a loosely restricted definition for our purpose. Take

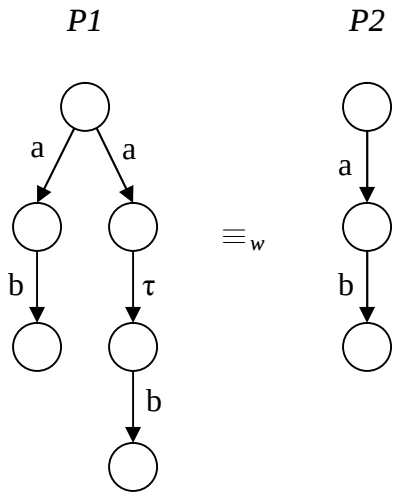


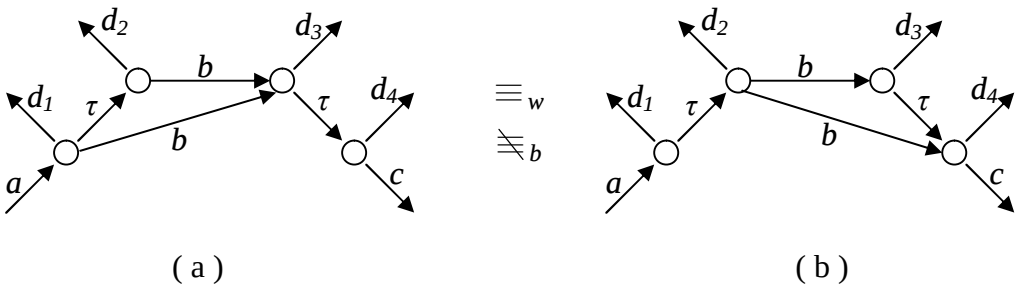
Figure 2.6. *P1* and *P2* are observation equivalent.

the popular definition of observation equivalence (weak bisimulation) in Figure 2.6 for example. The two LTSs, *P1* and *P2*, are behavioral equivalent according to the definition of observation equivalence. It is obvious to show that *P2* has less state number than *P1* does, but they are behavioral equivalent. So it is useful for model checking to take *P2* instead of *P1* to alleviate state explosion problem.

Although we can use the equivalence definition to minimize the model as smaller as we want, we can not forget the final objective for verification – checking the desired properties. If some features of original model would be lost after transforming to a smaller model, the checking technique can not provide any useful advice. Van Glabbeek et al. in [24] have proposed a suitable equivalence definition, called *branching bisimulation*, for verifying the correctness of concurrent systems. We present more details about it in the next section.

2.4 Branching Bisimulation

Branching bisimulation equivalence is introduced by Van Glabbeek and Weijland in [24]. They argue that this equivalence relation applied on labeled transition systems is finer than the observation equivalence of Milner [4]. Although branching bisimulation and observation equivalence are the same for a large class of processes [24], we prefer branching bisimulation for several reasons described below.



(a) (b)

Figure 2.7. Observation equivalence, but not branching bisimulation.

In the view of algorithm, branching bisimulation can be decided in $\boldsymbol{O}(mn)$ time complexity, where m is the number of transitions and n the number of states, and in $\boldsymbol{O}(m)$ space complexity [26][27]. Nevertheless, the fastest algorithm of observation equivalence has $\boldsymbol{O}(l \cdot n^{2.376})$ time complexity and $\boldsymbol{O}(n + m^{+})$ space complexity, where l is the number of actions and m^{+} is the number of edges after taking τ^{+} the transitive closure [24]. Therefore, we choose to implement branching bisimulation for better performance.

Abstracting the observable behaviors of processes is an important notion to determine whether two processes are behaviorally equivalent. Branching bisimulation preserves the branching structure, whereas the observation equivalence does not. Figure 2.7 extracted from [24] is an example to show that (a) and (b) are observation equivalent, but not for branching bisimulation.

Moreover, liveness properties are one of the most important correctness for model checking. Most liveness properties, such as progress, may use *eventually operator* to describe its requirement, but Van Glabbeek in [24] has shown that

observation equivalence is not adequate for a modal logic with eventually operator. Therefore, if we apply the observation equivalence to minimize a process, the liveness properties will be lost. However, branching bisimulation preserves liveness properties. Take Figure 2.8 for example. Consider the situation of composing *A* and *B* into $A \parallel B$. And then minimize it by observation equivalence and branching bisimulation. In Figure 2.8, (a) shows the result of observation equivalence minimization, and (b) is the result of branching bisimulation minimization. If we want to check the progress property of "always eventually *b*", it's hard to back trace on (a) to analyze. But in (b) there still remains the progress structure.

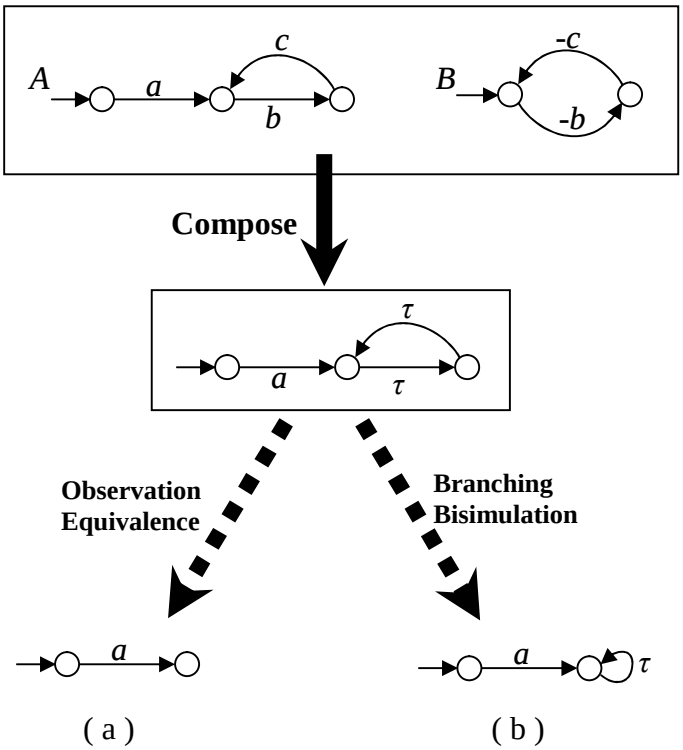


Figure 2.8. Branching Bisimulation preserves liveness.

Because branching bisimulation preserves more precise observable behaviors than

observation equivalence does and holds many nice features described in [24], we'd like to integrate it into compositional analysis to facilitate more efficiency.

Here's the formal *definition of branching bisimulation*. Let $LTS_1 = < S_1, A_1, \rightarrow_1, q_1 >$ and $LTS_2 = < S_2, A_2, \rightarrow_2, q_2 >$ be two labeled transition systems. $LTS_1 \equiv_b LTS_2$, where $\equiv_b$ denotes a branching bisimulation equivalence, if there exists a *binary symmetric relation* $\boldsymbol{R} \subseteq S_1 \times S_2$ such that

1. $q_1 \boldsymbol{R} q_2$

2. $\forall s, s'$: $s \boldsymbol{R} s'$, $\forall a \in A \cup \{\tau\}$, if $s \xrightarrow{a} t$, then

   - either $a = \tau \ \wedge \ t \boldsymbol{R} s'$, ····· Figure 2.9 (a)

   - or $\exists t'$, $\exists p_0, p_1, \ldots, p_n, \exists q_0, q_1, \ldots, q_m$,
   $s' = p_0 \xrightarrow{t} p_1 \cdots \xrightarrow{t} p_n = t_1 \xrightarrow{a} t_2 = q_0 \xrightarrow{t} q_1 \cdots \xrightarrow{t} q_m = t'$ s.t.
   $t \boldsymbol{R} t' \ \wedge \ s \boldsymbol{R} p_i \wedge t \boldsymbol{R} q_j$, $\forall i, j, 0 \le i \le n, 0 \le j \le m$. ····· Figure 2.9 (b)

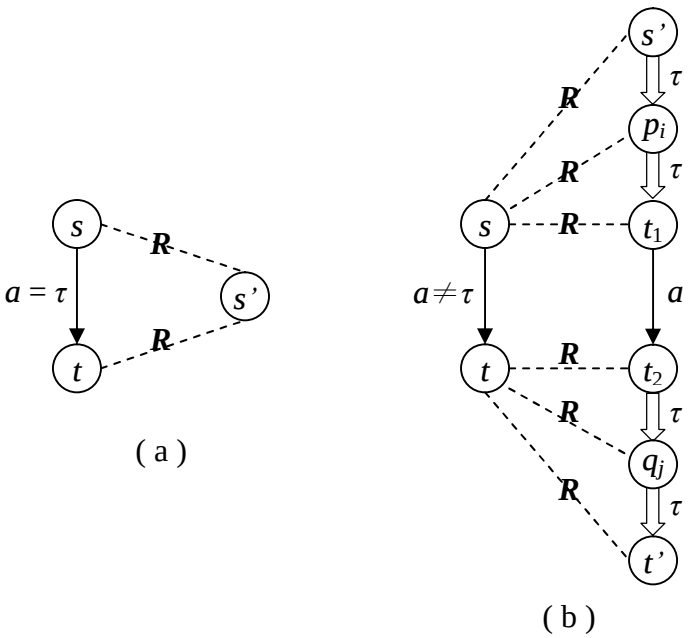


Figure 2.9. Definition of Branching Bisimulation

In Figure 2.9, we give a picture to sketch the relationship. The dashed line labeled by ***R*** indicates that it requires the relation between linked states. The hollow arrows in (b) correspond to $(\xrightarrow{t})^n$, where *n* is equal to or greater than 0.

2.5 Relational Coarsest Partition Problem[1]

In this section we discuss the general algorithm for *Relational Coarsest Partition Problem* (RCP for short). Kanellakis and Smolka [30] studied this problem to provide a solution on deciding equivalences over Labeled Transition Systems. The algorithm computes equivalence classes over states in an LTS, and refines them into different blocks with appropriate instability notion until all pairs of blocks are stable. Below we describe the algorithm based on [27] in Figure 2.10.

```
1:  Π := Π0
2:  While ( Π is not stable )
3:  begin
4:        Find ( B, B' ) unstable
5:        Π := Ref_Π( B, B' )
6:  end
```

Figure 2.10. The general algorithm for RCP problem.

Definition 2.1 Partitions and Blocks

Let *S* be a set of states of LTS. The collection $\Pi = \{ B_i \subseteq S \mid i \in \boldsymbol{I} \}$ is a *partition* of *S* if and only if $\bigcup_{i \in I} B_i = S$ and for $j \neq i$: $B_i \cap B_j = \phi$. The elements of a partition are called *blocks*. If Π and Π' are two partitions of *S*, then we say Π' *refines* Π if and only if $\forall B' \in \Pi'$, $\exists B \in \Pi$ such that $B' \subseteq B$. □

[1] Part of this section is extracted from Bouali92 [27].

At the beginning, the algorithm initializes a partition Π to Π_0, a partition with the one element of whole LTS. Then it repeats a ***refinement step*** which consists in finding an ***unstable*** pair of blocks and refining the current partition with respect to it, until a stable partition is obtained. With applying some instability notion related to equivalences, the algorithm computes a partition over LTS such that each block in the partition corresponds exactly to an equivalence class. **Definition 2.2** is the instability notion for branching bisimulation equivalence.

Definition 2.2 Instability Notion for Branching Bisimulation

Given an action a in A of LTS, a pair of blocks (B, B') of Π is ***a*-unstable** ***iff*** in the setting of ($B \neq B$' or $a \neq \tau$) and $f \neq \mathbf{Pos}_a(B,B') \neq B$, where

$$\mathbf{Pos}_a(B,B') = \{\, s \in B \mid \exists\, s_1, s_2, \ldots, s_n, s' : s_0 = s, \\ \forall\, i>0,\ s_i \in B \wedge s_{i-1} \xrightarrow{\ t\ } s_i,\ s_n \xrightarrow{\ a\ } s' \wedge s' \in B' \,\}. \qquad \square$$

The notion, ($B \neq B$' or $a \neq \tau$), means that we can not use ***τ-action*** to check the instability of B itself, i.e. (B, B). And the notion, $f \neq \mathbf{Pos}_a(B,B') \neq B$, means that only ***a part of*** the states in B reach states in B' via action a. If both notions are held, then the pair of blocks (B, B') is considered a-unstable.

A pair of blocks is said ***unstable*** in the setting described above, ***iff*** there exists an action a for which these blocks are a-unstable, otherwise it is said ***stable***. More generally, a block B of Π is said stable ***iff*** for all block B' in Π, the pair (B,B') is

stable. The partition Π is stable *iff* each of its block is stable. Otherwise, it is unstable.

When the algorithm finds an unstable pair of blocks, it applies **Definition 2.3** to refine the current partition with respect to it.

Definition 2.3 Refinement

If the pair of blocks (B, B') of Π is ***a*-unstable**, then a *refinement* $\boldsymbol{Ref}_{\Pi}^{a}(B, B')$ of Π is obtained by replacing B with the blocks $\boldsymbol{Pos}_a(B,B')$ and B- $\boldsymbol{Pos}_a(B,B')$. □

In [26], Groote and Vaandrager have shown that the analysis of instability can be essentially done by the use of $\boldsymbol{bottom}(B)$. It is formally defined as

$$\boldsymbol{bottom}(B) = \{ s \in B \mid s \xrightarrow{t} s' \Rightarrow s' \notin B \},$$

which is the subset of states in B those have no τ -transition ending in B. With the definition of $\boldsymbol{bottom}(B)$, the states in a block is separated into two kinds of set:

1. ***Non-bottom***: those states having τ-transitions ending in B, except self-τ-cycles.

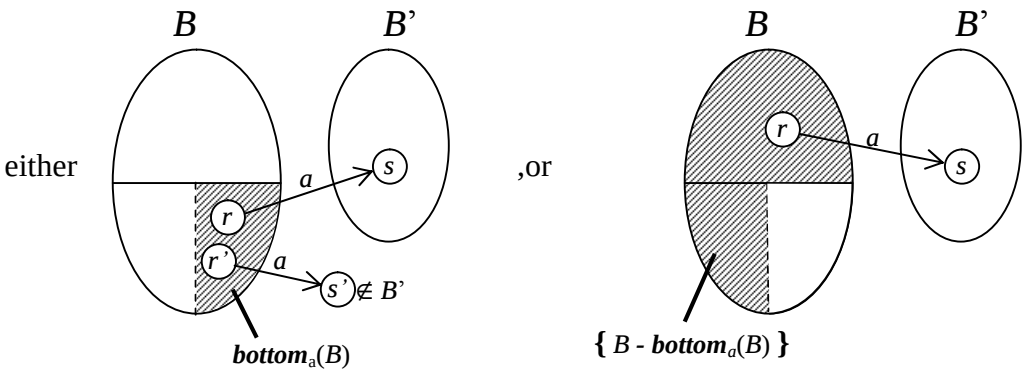
2. ***Bottom***: other states in B which do not belong in ***Non-bottom*** set.

In fact a pair of blocks (B, B') of Π is ***a*-unstable** *iff*

- either, ($\boldsymbol{bottom}_a(B)$, B') is a-unstable, where formally
  $\boldsymbol{bottom}_a(B)=\{r \in \boldsymbol{bottom}(B) \mid \exists s,\ r \xrightarrow{a} s\}$,

- or, $\boldsymbol{Pos}_a(\boldsymbol{bottom}_a(B), B') = f \wedge \exists r \in \{B\text{-}\boldsymbol{bottom}_a(B) \},\ \exists s \in B'$ s.t. $r \xrightarrow{a} s$

We give a picture to sketch the meaning described above in Figure 2.11. The oval represents a block, and the cross line in the middle of an oval stands for the separation of *bottom states* and *non-bottom states*.



(a) (b)

Figure 2.11. *a*-unstable of a pair (*B*, *B*').

In the case of (a), the dashed line takes apart the $\boldsymbol{bottom}_a(B)$ from the others in $\boldsymbol{bottom}(B)$. And there must exists two conditions at the same time, (1) some $r \in \boldsymbol{bottom}_a(B) \wedge s \in B'$: $r \xrightarrow{a} s$, and (2) some $r' \in \boldsymbol{bottom}_a(B) \wedge s' \notin B'$: $r' \xrightarrow{a} s'$, which means only a *part* of the states in $\boldsymbol{bottom}_a(B)$ go to block *B*' via action *a*.

In the case of (b), if $\boldsymbol{bottom}_a(B)$ is an empty set and there are other states in *non-bottom* set of *B* go to block *B*' via action *a*, then the pair of blocks (*B*, *B*') is also considered as *a*-unstable.

In [26], *B*' is also called a *splitter* by which *B* is split. In other words, a block which can not be split by other blocks in the current partition is shown stable.

Here we have reviewed all definitions of the algorithm in a brief view. A complete and detail algorithm for implementation will present in next chapter, and show some problems that may be encountered at design with viable solutions.