Contents

Chapter 8

Hierarchical Verification

In hierarchical design, we construct various models of a system at different levels of detail. The key verification issue, then, is to check that a detailed model P of the system conforms with a more abstract model Q. If both P and Q are reactive modules, then our notion of conformance requires that every finite sequence of observations $\overline{a}$ that may result from executing the detailed module P may also result from executing the more abstract module Q. In this case, we say that the module P *implements* the module Q.

8.1 Implementation of Reactive Modules

Given a reactive module P, an *initialized trace* of P is an initialized trace of the observation structure K_P, and the *language* L_P of the module P is the language L_{K_P} of the corresponding observation structure. The interaction of a module P with its environment is completely determined by the set $\mathsf{intf}X_P$ of interface variables, the set $\mathsf{extl}X_P$ of external variables, the awaits dependencies among the observable variables, and the set L_P of traces. Two modules that agree on these components will interact with other modules in the same way irrespective of their branching structures. This is illustrated by the following proposition which asserts that the traces of a compound module are completely determined by the traces of its components. In particular, if P and Q have identical observations then $L_{P \| Q}$ equals $L_P \cap L_Q$.

Proposition 8.1 [Traces of compound modules] *Let P and Q be compatible modules, and let $\overline{a}$ be a word over the observations of the compound module $P \| Q$. Then, $\overline{a}$ belongs to the language $L_{P \| Q}$ iff the projection $\mathsf{obs}X_P[\overline{a}]$ belongs to L_P and the projection $\mathsf{obs}X_Q[\overline{a}]$ belongs to L_Q.*

Exercise 8.1 {t2} [Traces of compound modules] Prove Proposition 8.1. ■

This leads to a natural way of comparing two modules.

IMPLEMENTATION

The reactive module P *implements* the reactive module Q, denoted $P \preceq^L Q$, if

1. every interface variable of Q is an interface variable of P: $\mathsf{intf}X_Q \subseteq \mathsf{intf}X_P$,
2. every external variable of Q is an observable variable of P: $\mathsf{extl}X_Q \subseteq \mathsf{obs}X_P$,
3. for all variables x in $\mathsf{obs}X_Q$ and y in $\mathsf{intf}X_Q$, if $y \prec_Q x$ then $y \prec_P x$, and
4. if $\overline{a}$ is an initialized trace of P then the projection $\overline{a}[\mathsf{obs}X_Q]$ of $\overline{a}$ onto the observable variables of Q is an initialized trace of Q.

The two modules P and Q are *trace equivalent*, denoted $P \simeq^L Q$, if $P \preceq^L Q$ and $Q \preceq^L P$.

Intuitively, if $P \preceq^L Q$ then the module P is *as complex as* the module Q: P has possibly more interface and external variables than Q, P has more await dependencies among its observable variables, and has less traces than Q, and thus, more constraints on its execution. The superscript L in the implementation relation $\preceq^L$ indicates that this relation is based on the languages of the modules.

Remark 8.1 [Implementation preorder] The implementation relation $\preceq^L$ on modules is reflexive and transitive. ■

Example 8.1 [Synchronous versus asynchronous mutual exclusion] The module *SyncMutex* of Figure 1.22 gives the synchronous solution to the mutual exclusion problem, and the module *Pete* of Figure 1.23 gives the asynchronous solution. Both the modules have identical interface variables, no external variables, and no await dependencies. Verify that every trace of *SyncMutex* is a trace of *Pete*, and thus, *SyncMutex* $\preceq^L$ *Pete*. However, the two modules are not trace equivalent. The word

$$(pc_1 = pc_2 = outC), (pc_1 = reqC, pc_2 = outC), (pc_1 = pc_2 = reqC)$$

is a trace of *Pete*, but is not a trace of *SyncMutex*. Intuitively, the asynchronous solution is more abstract, and admits more traces. ■

Example 8.2 [Nondeterministic versus deterministic scheduling] Recall the module *Scheduler* from Figure 1.4 consisting of atoms $A3$, $A4$, and $A5$. Consider the atom $A6$

```
$A6$: atom controls $proc$ reads $task_1, task_2$
  update
    $[\!]$ $task_1 = 0 \wedge task_2 = 0 \rightarrow proc' := 0$
    $[\!]$ $task_2 > 0 \rightarrow proc' := 2$
    $[\!]$ $task_1 > 0 \rightarrow proc' := 1$
```

When both the tasks are pending the atom $A6$ assigns the processor to one of them in a nondeterministic fashion. The module *NonDetScheduler* is like *Scheduler* with the atom $A5$ replaced by the atom $A6$ (*NonDetScheduler* no longer needs the private variable *prior*). The two modules *Scheduler* and *NonDetScheduler* have identical interface and external variables, and identical awaits dependencies among their observable variables. Verify that every trace of *Scheduler* is a trace of *NonDetScheduler*, but not vice versa. We have, *Scheduler* $\preceq^L$ *NonDetScheduler*. The module *Scheduler* is an implementation of the specification *NonDetScheduler*. The specification only requires that if one of the tasks is pending, then the processor should be assigned to a pending task, but does not specify a policy to resolve the contention when both tasks are pending. The implementation refines the specification by implementing a deterministic policy using the variable *prior*. ■

Example 8.3 [Binary counter specification] Recall the example of the sequential circuit for three-bit binary counter from Example 1.16. The counter takes two boolean inputs, represented by the external variables *start* and *inc*, for starting and incrementing the counter. The counter value ranges from 0 to 7, and is represented by three interface binary variables out_0, out_1, and out_2. The *specification* of the counter is the module *Sync3BitCounterSpec* of Figure 8.1. The module *Sync3BitCounter* of Figure 1.19 is a possible implementation. The correctness of the design *Sync3BitCounter* with respect to the specification *Sync3BitCounterSpec* is expressed by the fact that the two modules are trace-equivalent. ■

Exercise 8.2 {P2} [Zero-delay vs. unit-delay vs. buffered vs. lossy squaring] Recall the definitions of the modules *SyncSquare*, *SyncSquare2*, *AsyncSquare*, and *LossyAsyncSquare* (see Example 1.10, and Example 1.11) all of which compute squares of input numbers. Which pairs of modules among these four modules are related by the implementation relation $\preceq^L$? ■

Exercise 8.3 {P2} [Synchronous vs. asynchronous message passing] Consider the module *SyncMsg* of Figure 1.25 for synchronous message passing, and the module *AsyncMsg* of Figure 1.30 for asynchronous message passing. Does *SyncMsg* implement *AsyncMsg*? Are the two modules trace equivalent? ■

```
module $Sync3BitCounterSpec$ is
  interface $out_0, out_1, out_2$
  external $start, inc$
  atom controls $out_0$ reads $out_0$ awaits $start, inc$
    update
      $\|$ $start' = 1$                  $\rightarrow out_0' := 0$
      $\|$ $start' = 0 \wedge inc' = 1$ $\rightarrow out_0' := \neg out_0$
  atom controls $out_1$ reads $out_0, out_1$ awaits $start, inc$
    update
      $\|$ $start' = 1$                  $\rightarrow out_1' := 0$
      $\|$ $start' = 0 \wedge inc' = 1$ $\rightarrow out_1' := out_0 \oplus out_1$
  atom controls $out_2$ reads $out_0, out_1, out_2$ awaits $start, inc$
    update
      $\|$ $start' = 1$                  $\rightarrow out_2' := 0$
      $\|$ $start' = 0 \wedge inc' = 1$ $\rightarrow out_2' := (out_0 \wedge out_1) \oplus out_2$
```

Figure 8.1: Specification of three-bit counter

Composing two modules using parallel composition creates a module that is more complex than its components, while hiding a variable creates a simpler module with less number of observable variables.

Proposition 8.2 [Module operations and implementation] *(1) For compatible reactive modules P and Q, $P \,\|\, Q \preceq^L P$. (2) For a variable x and a reactive module P, $P \preceq^L$ **hide** x **in** P.*

Exercise 8.4 {T1} [Module operations and implementation] Prove Proposition 8.2. ■

The implementation problem

The implementation problem asks whether one module implements another module.

THE IMPLEMENTATION PROBLEM

An instance (P, Q) of the *implementation problem* consists of two reactive modules P and Q. The answer to the implementation problem (P, Q) is YES if P implements Q, and otherwise NO.

Recall that the module P implments the module Q if (1) $\mathsf{intf}X_Q \subseteq \mathsf{intf}X_P$, (2) $\mathsf{extl}X_Q \subseteq \mathsf{extl}X_P$, (3) for all x in $\mathsf{obs}X_Q$ and y in $\mathsf{intf}X_Q$, if $y \prec_Q x$ then $y \prec_P x$, and (4) if $\overline{a}$ is a trace of P then $\overline{a}[\mathsf{obs}X_Q]$ is a trace of Q. Checking first three conditions is easy, and the fourth condition reduces to the language inclusion problem. Thus, the implementation problem relates to the language-inclusion problem in the same way in which the invariant-verification problem relates to the reachability problem, and in which the STL verification problem relates to the STL model-checking problem.

Consider two modules P and Q such that the first three conditions for P to implement Q are met. Let $P' = \textbf{hide}\ (\mathsf{obs}X_P \backslash \mathsf{obs}X_Q)\ \textbf{in}\ P$. Then P' and Q have identical observable variables. The module P implements Q if every trace of P' is also a trace of Q, that is, if the answer to the language-inclusion problem $(K_{P'}, K_Q)$ is YES. The complexity of solving the language-inclusion question is exponential in its second argument. The next theorem concerning checking implementation relation between two propositional modules follows.

Theorem 8.1 [The implementation problem] *Let P be a propositional module with k propositional variables and let Q be a propositional module with ℓ propositional variables. The propositional implementation problem (P, Q) can be solved using the language-inclusion algorithm in time $O(4^k \cdot 2^{2^\ell})$.*

Remark 8.2 [Space complexity of implementation problem] The propositional implementation problem (P, Q) is EXPSPACE hard in its second argument. Checking implementation requires searching the product of the observation structure of P and the determinization of the the observation structure of Q, and can be performed in space $O(k \cdot 2^\ell)$ if P has k variables and Q has ℓ variables. It follows that propositional implementation problem is complete for EXPSPACE. ■

If Q is an observably-deterministic module, then so is the observation structure K_Q. In this case, the language inclusion question $(K_{P'}, K_Q)$ can be solved without determinization.

Theorem 8.2 [Deterministic case of implementation problem] *Let P be a propositional module with k propositional variables and let Q be a propositional observably-deterministic module with ℓ propositional variables. The propositional implementation problem (P, Q) can be solved in time $O(4^{k+\ell})$.*

Remark 8.3 [Deterministic case of space complexity of implementation proble] If Q is observably-deterministic, then the propositional implementation problem (P, Q) is PSPACE-complete. ■

8.1.1 From Bisimilarity to Implementation

Given two observation structures K_1 and K_2, because the language-inclusion problem (K_1, K_2) is hard, in practice, it is important to find sufficient conditions for $L_{K_1} \subseteq L_{K_2}$ that can be checked efficiently. One such sufficient condition is bisimilarity.

State preorders

A *state preorder* $\preceq$ is a family of preorders, one preorder $\preceq_K$ on the of each observation structure K. The *trace preorder* $\preceq^L$ is the following state preorder: for two states s and t of an observation structure K, let $s \preceq^L_K t$ iff $L_K(s) \subseteq L_K(t)$.

A state preorder allows us to compare states. To compare two observation structures with identical observations using a state preorder, we consider the disjoint union of the two structures, and check if every initial state of one is related to some initial state of the other. It also leads to a way of comparing two reactive modules.

STRUCTURE AND MODULE PREORDERS OF A STATE PREORDER

Let $\preceq$ be a state preorder. Let K_1 and K_2 be two disjoint observation structures. Let σ^I_1 be the initial region of K_1, and let σ^I_2 be the initial region of K_2. Then, $K_1 \preceq K_2$ if for all states $s \in \sigma^I_1$, there is a state $t \in \sigma^I_2$ such that $s \preceq_{K_1+K_2} t$.

For two modules P and Q, $P \preceq Q$ if (1) every interface variable of Q is an interface variable of P, (2) every external variable of Q is an observable variable of P, (3) for all variables x in $\mathsf{obs}X_Q$ and y in $\mathsf{intf}X_Q$, if $y \prec_Q x$ then $y \prec_P x$, and (4) $K_{P'} \preceq K_Q$ for $P' = \textbf{hide}\ (\mathsf{obs}X_P \backslash \mathsf{obs}X_Q)\ \textbf{in}\ P$.

Observe that the structure preorder relates two structures only when they have identical observations, and hence, when comparing two modules we use hiding before we compare the corresponding observation structures. For the bisimilarity relation, $K_1 \preceq^B K_2$ means that every initial state of K_1 is bisimilar to some initial state of K_2.

Remark 8.4 [Bisimulation preorder] Since bisimilarity is an equivalence relation over states, if $K_1 \preceq^B K_2$ and both structures have unique initial states, then $K_2 \preceq^B K_1$. If the two modules P and Q have identical interface and external variables, and unique initial states, then if $P \preceq^B Q$ then $Q \preceq^B P$. ■

Exercise 8.5 {`T1`} [Checking bisimilarity preorder] Given two observation structures K_1 and K_2, what is the time complexity of checking $K_1 \preceq^B K_2$? ■

For the trace preorder, the induced preorder over observation structures is language inclusion, and the induced preorder over modules is implementation. Since bisimilar states have identical languages, proving bisimilarity preorder is a sufficient condition for proving implementation. However, it is not a necessary condition, because bisimilarity is more distinguishing than language equivalence.

Proposition 8.3 [Trace and bisimilarity preorders] *For two observation structures K_1 and K_2, if $K_1 \preceq^B K_2$, then $K_1 \preceq^L K_2$. For two reactive modules P and Q, if $P \preceq^B Q$ then $P \preceq^L Q$.*

Example 8.4 [Trace and bisimilarity preorders] In Example 8.1, we noted that *SyncMutex* $\preceq^L$ *Pete*. However, *SyncMutex* $\preceq^B$ *Pete* does not hold.

In Example 8.2, we noted that *Scheduler* $\preceq^L$ *NonDetScheduler*. Hopwever, *Scheduler* $\preceq^B$ *NonDetScheduler* does not hold.

In Example 8.3, we noted that *Sync3BitCounter* and *Sync3BitCounterSpec* are trace-equivalent. Verify that the two modules are equivalent according to the bisimilarity preorder also. ■

8.2 Compositional Reasoning

8.2.1 Compositionality

If we prove that a module P implements another module Q, can we substitute P for Q in all contexts? Compositional proof rules admit such deductions, thereby reducing reasoning about compound modules to reasoning about the component modules.

COMPOSITIONALITY

The preorder $\preceq$ on reactive modules is *compositional* if for all modules P and Q, if $P \preceq Q$ then

1. for every reactive module R that is compatible with P, R is compatible with Q and $P \| R \preceq Q \| R$;
2. for variable x of P, **hide** x **in** $P \preceq$ **hide** x **in** Q;
3. for every variable renaming ρ, $P[\rho] \preceq Q[\rho]$.

A compositional equivalence on modules is called a *module congruence*.

Remark 8.5 [Congruence] If $\preceq$ is a compositional preorder on modules, then the symmetric closure of $\preceq$ is a module congruence. ■

Theorem 8.3 [Compositionality of implementation] *The implementation preorder $\preceq^L$ on modules is compositional.*

Proof. Consider two reactive modules P and Q such that $P \preceq^L Q$. The cases corresponding to the operations of hiding and renaming are straightforward.

We consider only parallel composition. Let R be a reactive module that is compatible with P.

First, we need to establish that Q and R are compatible. Since $\mathsf{intf}X_P \cap \mathsf{intf}X_R$ is empty, and $\mathsf{intf}X_Q \subseteq \mathsf{intf}X_P$, we conclude that $\mathsf{intf}X_Q \cap \mathsf{intf}X_R$ is empty. Asymmetricity of $(\prec_Q \cup \prec_R)^+$ follows from (1) $(\prec_P \cup \prec_R)^+$ is asymmetric, and (2) for two variables $x, y \in \mathsf{obs}X_Q$, if $x \prec_Q y$ then $x, y \in \mathsf{obs}X_P$ with $x \prec_P y$.

Next, we establish that every initialized trace of $P \| R$ is also an initialized trace of $Q \| R$. Let $\overline{a}$ be an initialized trace of $P \| R$. From Proposition 8.1, $\mathsf{obs}X_P[\overline{a}]$ is an initialized trace of P and $\mathsf{obs}X_R[\overline{a}]$ is an initialized trace of R. Since $P \preceq^L Q$, $\mathsf{obs}X_Q[\overline{a}]$ is an initialized trace of Q. Again, from Proposition 8.1, $\mathsf{obs}X_{Q \| R}[\overline{a}]$ is an initialized trace of $Q \| R$. ■

Corollary 8.1 [Congruence of trace equivalence] *Trace equivalence is a module congruence.*

Suppose that we wish to prove that a compound module $P_1 \| P_2$ implements the abstraction $Q_1 \| Q_2$, where Q_1 is an abstraction of P_1 and Q_2 is an abstraction of P_2. By Theorem 8.3, it suffices to prove separately that (1) the component module P_1 implements Q_1, and (2) the component module P_2 implements Q_2. Both proof obligations (1) and (2) involve smaller state spaces than the original proof obligation.

Example 8.5 [Compositional proof] Consider the synchronous message-passing protocol

$$\mathit{SyncMsg} \;=\; \textbf{hide } \mathit{ready}, \mathit{transmit}, \mathit{msg}_S \textbf{ in } \mathit{SyncSender} \,\|\, \mathit{Receiver}$$

and the asynchronous message passing protocol

$$\mathit{AsyncMsg} \;=\; \textbf{hide } \mathit{ready}, \mathit{transmit}, \mathit{msg}_S \textbf{ in } \mathit{AsyncSender} \,\|\, \mathit{Receiver}.$$

Suppose we wish to establish that the synchronous protocol is an implementation of the asynchronous one:

$$\mathit{SyncMsg} \;\preceq^L\; \mathit{AsyncMsg}.$$

Then, since $\preceq^L$ is compositional, it suffices to establish that

$$\mathit{SyncSender} \;\preceq^L\; \mathit{AsyncSender}.$$

Verify that synchronous sender *SyncSender* indeed is an implementation of the asynchronous sender *AsyncSender*. ■

Exercise 8.6 {`T3`} [Bisimilarity congruence] Prove that the bisimulation preorder $\preceq^B$ is compositional. ■

It also follows that the state logic SAL is compositional:

Corollary 8.2 [Compositionality of SAL] *If a reactive module P satisfies an* SAL *formula ϕ, then every compound module $P \| Q$ also satisfies the* SAL *formula ϕ.*

8.2.2 Assume-guarantee Reasoning

Compositional proof rules, while useful, may not always be applicable. In particular, P_1 may not implement Q_1 for all environments, but only if the environment behaves like P_2, and vice versa. In this case, *assumption-guarantee proof rules* are needed. An assume-guarantee proof rule asserts that in order to prove that $P_1 \| P_2$ implements $Q_1 \| Q_2$, it suffices to prove (1) $P_1 \| Q_2$ implements Q_1, and (2) $Q_1 \| P_2$ implements Q_2. Both proof obligations (1) and (2) typically involve smaller state spaces than the original proof obligation, because the compound module $P_1 \| P_2$ usually has the largest state space involved. Observe the circular nature of the assume-guarantee reasoning. Its correctness depends crucially on the fact that a module does not constrain the behavior of its environment, and thus, interacts with other modules in a non-blocking way.

Theorem 8.4 [Assume-guarantuee reasoning] *Let P_1 and P_2 be two compatible reactive modules, and let Q_1 and Q_2 be two compatible reactive modules. If $P_1 \| Q_2 \preceq^L Q_1$, $Q_1 \| P_2 \preceq^L Q_2$, and every external variable of $Q_1 \| Q_2$ is an observable variable of $P_1 \| P_2$, then $P_1 \| P_2 \preceq^L Q_1 \| Q_2$.*

Proof. Consider four modules P_1, P_2, Q_1, and Q_2 such that

1. P_1 and P_2 are compatible,
2. Q_1 and Q_2 are compatible,
3. every external variable of $Q_1 \| Q_2$ is an observable variable of $P_1 \| P_2$,
4. $P_1 \| Q_2 \preceq^L Q_1$, and
5. $Q_1 \| P_2 \preceq^L Q_2$.

We wish to establish that $P_1 \| P_2 \preceq^L Q_1 \| Q_2$. The definition of implementation has four requirements. Let us consider these four goals one by one.

Goal 1: To show that every interface variable of $Q_1 \| Q_2$ is an interface variable of $P_1 \| P_2$, let x be an interface variable of $Q_1 \| Q_2$. Due to symmetry, it suffices to consider the case that x is an interface variable of Q_1. By assumption (4), x is an interface variable of $P_1 \| Q_2$. The assumption (2) implies that x is not an interface variable of Q_2. It follows, from the definition of parallel composition, that x is an interface variable of P_1, and hence, of $P_1 \| P_2$.

Goal 2: The second requirement that every external variable of $Q_1 \| Q_2$ is an observable variable of $P_1 \| P_2$ is assumption (3).

Goal 3: We wish to show that if $y \prec_{Q_1 \| Q_2} x$ then $y \prec_{P_1 \| P_2} x$. Since $\prec_{Q_1 \| Q_2}$ is the transitive closure of the union $\prec_{Q_1}$ and $\prec_{Q_2}$, and by symmetry, it suffices to prove that if $y \prec_{Q_1} x$ then $y \prec_{P_1 \| P_2} x$.

Consider an interface variable y and an observable variable of x of Q_1 such that $y \prec_{Q_1} x$. From assumption (4), we have $y \prec_{P_1\|Q_2} x$. We know that $\prec_{P_1\|Q_2}$ is the transitive closure of the union of $\prec_{P_1}$ and $\prec_{Q_2}$. Hence, whenever $y \prec_{Q_1} x$, there is a finite chain of awaits dependencies such that

$$y \prec_{P_1} y_1 \prec_{Q_2} y_2 \prec_{P_1} \cdots \; x \quad (\dagger)$$

Similarly, if $y \prec_{Q_2} x$, then there exists a chain of awaits dependencies

$$y \prec_{P_2} y_1 \prec_{Q_1} y_2 \prec_{P_2} \cdots \; x \quad (\ddagger)$$

By repeatedly applying (†) and (‡), since the awaits relations are acyclic, and the number of variables is finite, if $y \prec_{Q_1} x$, then there exists a finite chain of awaits dependencies

$$y \prec_{P_1} y_1 \prec_{P_2} y_2 \prec_{P_1} \cdots \; x$$

and thus, $y \prec_{P_1\|P_2} x$.

Goal 4: We wish to establish that every trace of $P_1\|P_2$ is also a trace of $Q_1\|Q_2$. We start by defining some additional concepts. For simplicity, in the following we omit explicit projections. For instance, if X is a superset of $\mathsf{obs}X_P$, and $\overline{s}$ is sequence of valuations for X such that $\overline{s}[\mathsf{obs}X_P]$ is a trace of P, then we consider $\overline{s}$ also to be a trace of P.

Given a module P, a subset $X \subseteq \mathsf{obs}X_P$ of the observable variables is *awaits-closed* if, whenever $y \prec_P x$ and $y \in X$, then $x \in X$. For an awaits-closed set X, the pair $(\overline{s}_{0\ldots m}, t)$ consisting of a trace $\overline{s}_{0\ldots m} \in L_P$ of P and a valuation t for X is said to be a X*-partial-trace* of P if there exists an observation s_{m+1} such that (1) $s_{m+1}[X] = t$, and (2) $\overline{s}_{0\ldots(m+1)} \in L_P$. Thus, partial-traces are obtained by executing only some of the subrounds of the last update round. The following facts about partial-traces follow from the definition of reactive modules.

1. If $P \preceq^L Q$ and X is awaits-closed for P, then X is awaits-closed for Q. If $P \preceq^L Q$, and $(\overline{s}, t)$ is a X-partial-trace of P, then $(\overline{s}, t)$ is a X-partial-trace of Q. Thus, inclusion of traces is equivalent to inclusion of partial traces.

2. The partial-traces of a compound module are determined from the partial-traces of the components: $(\overline{s}, t)$ is a X-partial-trace of $P\|Q$ iff it is a X-partial-trace of both P and Q.

3. If $(\overline{s}, t)$ is a X-partial-trace of P, and u is a valuation for a subset Y of the external variables of P, then $(\overline{s}, t \cup u)$ is a $(X \cup Y)$-partial-trace of P. This property is due the nonblocking nature of reactive modules.

Let $X_1, \ldots X_k$ be a partitioning of $\mathsf{obs}X_{P_1\|P_2}$ into disjoint subsets such that (1) each X_i either contains only external variables of $P_1\|P_2$ or contains only interface variables of P_1 or only interface variables of P_2, and (2) $x \prec_{P_1\|P_2} y$

and $x \in X_i$ then $y \in X_j$ for some $j < i$. Let $Y_0 = \emptyset$, and for $0 \leq i < k$, $Y_{i+1} = Y_i \cup X_i$. Each such set Y_i is awaits-closed. Let $\mathcal{L}$ be the set of pairs $(\overline{s}, t)$ such that $(\overline{s}, t)$ is a X-partial-trace of $P_1 \| P_2$ for $X = Y_j$ for some $0 \leq j \leq k$. Define an ordering $<$ over $\mathcal{L}$: if $(\overline{s}, t)$ is a Y_j-partial-trace with $j < k$, and $(\overline{s}, u)$ is a Y_{j+1}-partial-trace with $u[Y_j] = t$ then $(\overline{s}, t) < (\overline{s}, u)$; and $(\overline{s}, t)$ is a Y_k-partial-trace then $(\overline{s}, t) < (\overline{s} \cdot t, \emptyset)$. Clearly, the ordering $<$ is well founded. By well-founded induction with respect to $<$, we prove that every partial-trace in $\mathcal{L}$ is a partial-trace of $Q_1 \| Q_2$.

Consider $(\overline{s}, \emptyset)$ in $\mathcal{L}$. If $\overline{s}$ is the empty trace, then $(\varepsilon, \emptyset)$ is a partial-trace of all modules. Otherwise, $\overline{s}$ is nonempty: $\overline{s} = \overline{t} \cdot u$. Then $(\overline{t}, u)$ is a Y_k-partial-trace of $P_1 \| P_2$. Since $(\overline{t}, u) < (\overline{s}, \emptyset)$, by induction hypothesis, $(\overline{t}, u)$ is a partial-trace of $Q_1 \| Q_2$, and hence, so is $(\overline{s}, \emptyset)$.

Consider $(\overline{s}, t)$ in $\mathcal{L}$ such that t is a valuation for Y_{j+1} for some $0 \leq j < k$. Let $u = t[Y_j]$. Then, $(\overline{s}, u) < (\overline{s}, t)$. By induction hypothesis, $(\overline{s}, u)$ is a Y_j-partial-trace of $Q_1 \| Q_2$. By the property (2) of partial-traces, $(\overline{s}, u)$ is a Y_j-partial-trace of both Q_1 and Q_2. Consider $Y_{j+1} = Y_j \cup X_j$. We know that X_j contains interface variables of at most one of P_1 and P_2. Without loss of generality, let us assume that X_j contains no interface variables of P_2, and hence, no interface variables of Q_2. By property (3) of partial-traces, the Y_j-partial-trace $(\overline{s}, u)$ of Q_2 can be extended with any valuation for X_j. In particular, $(\overline{s}, t)$ is a Y_{j+1}-partial-trace of Q_2. Hence, $(\overline{s}, t)$ is a Y_{j+1}-partial-trace of $P_1 \| Q_2$. Since $P_1 \| Q_2 \preceq^L Q_1$, and by property (1) of partial-traces, $(\overline{s}, t)$ is a partial-trace of Q_1. Again, by property (2) of partial-traces, $(\overline{s}, t)$ is a partial-trace of $Q_1 \| Q_2$. ∎

Example 8.6 [Assume guarantee reasoning] To illustrate the application of assume guarantee reasoning, we consider a simple version of the alternating-bit protocol. The sender process is the module *ABPSender* of Figure 8.2. The private variable x indicates the bit to be sent with the next message. The message is transmitted by issuing the interface event $\mathit{transmit}_S$, and updating the variables *abp* and *msg* to the message contents. The acknowledgements issued by the receiver are stored in the private buffer z. After sending the message, the process removes an acknowledgement from z. If the acknowledgement equals the current value of the alternating-bit x, the sender concludes a correct delivery of the message, and updates the alternating-bit x.

The receiver process *ABPReceiver* is symmetric, and is shown in Figure 8.3 The messages received from the sender are stored in the private buffer z (for simplicity, the message is ignored, and the alternating-bit is stored). The process removes the first message in z in the variable x, which is, then, issued at a later time along with the interface event $\mathit{transmit}_R$.

Consider the module $\mathit{ABP} = \mathit{ABPSender} \,\|\, \mathit{ABPReceiver}$. The observable behavior of *ABP* is very regular: first the sender issues $\mathit{transmit}_S$ with the bit 0,

```
module $ABPSender$ is
  interface $transmit_S\colon \mathbb{E}$;  $abp\colon \mathbb{B}$;  $msg\colon \mathbb{M}$
  external $transmit_R\colon \mathbb{E}$;  $ack\colon \mathbb{B}$
  private $consume\colon \mathbb{E}$;  $pc\colon \{send, wait\}$;  $x\colon \mathbb{B}$;  $z\colon$ queue of $\mathbb{B}$
  passive atom
    controls $z$
    reads $consume, transmit_R$
    awaits $consume, transmit_R, ack$
    init
      $\| \; true \rightarrow z' := EmptyQueue$
    update
      $\| \; consume? \wedge transmit_R? \quad \rightarrow z' := Enqueue(ack', Dequeue(z))$
      $\| \; consume? \wedge \neg transmit_R? \rightarrow z' := Dequeue(z)$
      $\| \; \neg consume? \wedge transmit_R? \rightarrow z' := Enqueue(ack', z)$
  lazy atom
    controls $consume, x$
    reads $pc, z, x, consume$
    init
      $\| \; true \rightarrow x' := 0$
    update
      $\| \; pc = wait \wedge z \neq EmptyQueue \wedge x = Front(z) \rightarrow consume!$
      $\| \; pc = wait \wedge z \neq EmptyQueue \wedge x \neq Front(z) \rightarrow consume!;\ x' := \neg x$
  passive atom
    controls $pc$
    reads $pc, consume, transmit_S$
    awaits $consume, transmit_S$
    init
      $\| \; true \rightarrow pc' := send$
    update
      $\| \; pc = send \wedge transmit_S? \rightarrow pc' := wait$
      $\| \; pc = wait \wedge consume? \quad \rightarrow pc' := send$
  lazy atom
    controls $transmit_S, msg, abp$
    reads $pc, transmit_S, x$
    update
      $\| \; pc = send \rightarrow transmit_S!;\ abp' := x;\ msg' := \mathbb{M}$
```

Figure 8.2: Sender process of Alternating-bit Protocol

then the receiver issues $transmit_R$ with the acknowledgement 0, then the sender issues $transmit_S$ with the bit 1, then the receiver issues $transmit_R$ with the acknowledgement 1.

Figure 8.4 shows simpler abstract versions of the sender and receiver. The module *AbstractSender* differs from the module *ABPSender* in two ways. It assumes that (1) it will always recieve the correct acknowledgements, and (2) the acknowledgement events issued by the receiver strictly alternate with the events issued by the sender. Consequently, it does not read the values of the acknowledgements, and it does not buffer the acknowledgements. The process *AbstractReceiver* is a similar simplification of *ABPReceiver*.

Suppose we wish to establish that

$$ABPSender \parallel ABPReceiver \preceq^L AbstractSender \parallel AbstractReceiver. \quad (\dagger)$$

Compositionality cannot simplify this goal, becuase neither *ABPSender* implement *AbstractSender*, nor does *ABPReceiver* implement *AbstractReceiver*. However, verify that both

$$ABPSender \parallel AbstractReceiver \preceq^L AbstractSender,$$

and

$$AbstractSender \parallel ABPReceiver \preceq^L AbstractReceiver$$

hold. Then, by assume-guarantee theorem we can conclude the obligation (†). ■

Exercise 8.7 {`T2`} [Side condition in assume-guarantee rule] Show that the assumption that every external variable of $Q_1 \| Q_2$ is an observable variable of $P_1 \| P_2$ is essential (i.e. it does not follow from the assumptions (1), (2), (4), and (5) in the proof of the assume-guarantee theorem. ■

Exercise 8.8 {`T3`} [Assume-guarantee for bisimilarity] Does Theorem 8.4 hold for the bisimulation preorder $\preceq^B$? ■

8.3 Simulation Relations

Establishing trace preorder between two observation structures is computationally hard. While bisimilarity of two structures is a sufficient condition to establish trace preorder, bisimilarity over structures in an equivalence relation, and does not admit the implementation to have less traces than the specification. Simulation relations offer a practical alternative: on one hand, computing simulation relations is computationally easier than establishing trace preorder,

```
**module** $ABPReceiver$ **is**
  **external** $transmit_S : \mathbb{E};\ abp : \mathbb{B};\ msg : \mathbb{M}$
  **interface** $transmit_R : \mathbb{E};\ ack : \mathbb{B}$
  **private** $consume : \mathbb{E};\ pc : \{send, wait\};\ x : \mathbb{B};\ z :$ **queue of** $\mathbb{B}$
  **passive atom**
    **controls** $z$
    **reads** $consume, transmit_S$
    **awaits** $consume, transmit_S, abp$
    **init**
      $\| \; true \rightarrow z' := EmptyQueue$
    **update**
      $\| \; consume? \wedge transmit_S? \rightarrow z' := Enqueue(abp', Dequeue(z))$
      $\| \; consume? \wedge \neg transmit_S? \rightarrow z' := Dequeue(z)$
      $\| \; \neg consume? \wedge transmit_S? \rightarrow z' := Enqueue(abp', z)$
  **lazy atom**
    **controls** $consume, x$
    **reads** $pc, z$
    **update**
      $\| \; pc = wait \wedge z \neq EmptyQueue \rightarrow consume!;\ x' := Front(z)$
  **passive atom**
    **controls** $pc$
    **reads** $pc, consume, transmit_R$
    **awaits** $consume, transmit_R$
    **init**
      $\| \; true \rightarrow pc' := wait$
    **update**
      $\| \; pc = send \wedge transmit_R? \rightarrow pc' := wait$
      $\| \; pc = wait \wedge consume? \rightarrow pc' := send$
  **lazy atom**
    **controls** $transmit_R, ack$
    **reads** $pc, transmit_R, x$
    **update**
      $\| \; pc = send \rightarrow transmit_R!;\ ack' := x$
```

Figure 8.3: Receiver process of Alternating-bit Protocol

```
module $AbstractSender$ is
  interface $transmit_S : \mathbb{E};\ abp : \mathbb{B}$
  external $transmit_R : \mathbb{E}$
  private $pc : \{send, wait\};\ x : \mathbb{B}$
  passive atom
    controls $pc$
    reads $pc, transmit_S, transmit_R$
    awaits $transmit_R, transmit_S$
    init
      $\| \ true \to pc' := send$
    update
      $\| \ pc = send \wedge transmit_S? \to pc' := wait$
      $\| \ pc = wait \wedge transmit_R? \to pc' := send$
  lazy atom
    controls $x, transmit_S, abp$
    reads $pc, x, transmit_S$
    init
      $\| \ true \to x' := 0$
    update
      $\| \ pc = send \to transmit_S!;\ abp' := x;\ x' := \neg x$

module $AbstractReceiver$ is
  external $transmit_S : \mathbb{E}$
  interface $transmit_R : \mathbb{E};\ ack : \mathbb{B}$
  private $pc : \{send, wait\};\ x : \mathbb{B}$
  passive atom
    controls $pc$
    reads $pc, transmit_S, transmit_R$
    awaits $transmit_R, transmit_S$
    init
      $\| \ true \to pc' := wait$
    update
      $\| \ pc = send \wedge transmit_R? \to pc' := wait$
      $\| \ pc = wait \wedge transmit_S? \to pc' := send$
  lazy atom
    controls $x, transmit_R, ack$
    reads $pc, x, transmit_R$
    init
      $\| \ true \to x' := 0$
    update
      $\| \ pc = send \to transmit_R!;\ ack' := x;\ x' := \neg x$
```

Figure 8.4: Abstract Sender and Receiver of Alternating-bit Protocol

while on the other hand, existence of simulation relations is much less stringent requirement compared to bisimilarity.

Simulation

Let $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$ be an observation structure. A *simulation* $\preceq$ of K is a binary relation on the state space such that for all states s and t of K, if $s \preceq t$ then (1) $\langle\!\langle s\rangle\!\rangle = \langle\!\langle t\rangle\!\rangle$ and (2) if $s \rightarrow s'$, then there is a state t' such that $t \rightarrow t'$ and $s' \preceq t'$. The state t *simulates* the state s, denoted $s \preceq^S_K t$, if there is a simulation $\preceq$ such that $s \preceq t$.

From the definition of simulation relations, it follows that the union of two simulation relations is also a simulation relation.

Proposition 8.4 [Union-closure of simulation relations] *Let K be an observation structure with state-space Σ. For two simulation relations $\preceq_1$ and $\preceq_2$, their union $\preceq_1 \cup \preceq_2$ is a simulation relation.*

It follows that the set of simulation relations forms a complete partial-order with respect to the subset relation.

Corollary 8.3 [Maximal simulation] *For an observation structure K, the relation $\preceq^S_K$ is a simulation of K, and equals the union of all simulation relations of K.*

The maximal simulation relation $\preceq^S_K$ is reflexive and transitive, and thus, a state preorder. This follows from the fact that reflexive-transitive closure a simulation is also a simulation.

Proposition 8.5 [Simulation preorder] *For an observation structure K, if $\preceq$ is a simulation of K, then so is its reflexive-stransitive closure $\preceq^*$.*

Exercise 8.9 {`T2`} [Simulation preorder] Prove Proposition 8.5. ■

Recall the alternative definitions of bisimilarity from Chapter 6. Similarity relation can be also be explained in various ways. Let us consider the similarity game on the graph of the observation structure K. Player I, the *protagonist*, attempts to show that the state t simulates the state s, while Player II, the *adversary*, tries to establish otherwise. If the two given states have different observations, then the adversary wins immediately. Throughout the game, each player has an active state. Initially, the active state of the adversary is s, and the active state of the protagonist is t. In each move of the game, the adversary replaces its active state by one of its successors, say s'; the protagonist, then, must replace its own active state with one of its successors t' such that s' and t' have identical observations. If the protagonist cannot find such a replacement, then the adversary wins the game. The state t simulates the state s iff the adversary does not have a winning strategy; that is, all of possible moves of the adversary can perpetually be matched by the protagonist. Contrast this game with the bisimilarity game: the bisimilarity game has two active states at each step, the adversary chooses one of the two active states, and replaces it by one of its successors, and the protagonist is required to find a replacement for the other active state. Thus, similarity game is like bisimilarity game in which, the adversary starts playing from s and never switches sides.

Example 8.7 [Simulation game] Let us revisit the bisimilarity game of Example 6.9 (See Figure 6.7). States s_0 and u_0 are bisimilar, and hence, similar. States s_0 and t_0 are not bisimilar. Now let us consider the similarity game. Suppose initially the active state of the adversary is s_0, while the active state of the protagonist is t_0. Verify that the adversary has a winning strategy in this case. Consequently, the state t_0 does *not* simulate the state s_0, and $s_0 \not\preceq^S t_0$. On the other hand, suppose initially the active state of the adversary is t_0 and the active state of the protagonist is s_0. In this case, the protagonist can match every move of the adversary. In fact, $\{(t_0, s_0), (t_1, s_1), (t_2, s_1), (t_3, s_2), (t_4, s_3)\}$ is a simulation relation. Consequently, the state s_0 *does* simulate the state t_0, and $t_0 \preceq^S s_0$. ■

Remark 8.6 [Simulation vs. bisimulation] Recall that an equivalence relation $\cong$ on the states of an observation structure K is a bisimulation iff whenever $s \cong t$, (1) $\langle\langle s \rangle\rangle = \langle\langle t \rangle\rangle$, and (2) if $s \to s'$, then there is a state t' such that $t \to t'$ and $s' \cong t'$. It follows that the bisimulations of K are precisely the symmetric simulations of K; that is, the equivalence $\cong$ on the states of K is a bisimulation iff both $\cong$ and $\cong^{-1}$ are simulations of K. ■

To prove that the language of a state s is included in the language of a state t, it suffices to prove that t simulates s.

Theorem 8.5 [Simulation vs. language inclusion] *Let s and t be two states of an observation structure K. If $s \preceq^S_K t$, then $s \preceq^L_K t$.*

Proof. Consider two states s and t of K such that $s \preceq^S_K t$. Consider a source-s trajectory $\overline{s}_{0\ldots m}$. Let $t_0 = t$. For $i = 1, \ldots m$, by induction on i, since $s_{i-1} \preceq^S_K t_{i-1}$ and $s_{i-1} \to s_i$, there exists a state t_i such that $s_i \preceq^S_K t_i$ and $t_{i-1} \to t_i$. Thus, $\overline{t}_{0\ldots m}$ is a source-t trajectory of K. Furthermore, for all $0 \leq i \leq m$, $\langle\langle s_i \rangle\rangle = \langle\langle t_i \rangle\rangle$, and hence, $\langle\langle \overline{s}_{0\ldots m} \rangle\rangle$ is also a source-t trace of K. ■

The simulation preorder allows comparing two observations structures: for two observation structures K_1 and K_2 with disjoint state-spaces and identical observations, $K_1 \preceq^S K_2$ if for every initial state s of K_1, there exists an initial state t of K_2 such that $s \preceq^S_{K_1+K_2} t$.

Corollary 8.4 [Simulation preorder vs. trace preorder] *For two observation structures K_1 and K_2, if $K_1 \preceq^S K_2$, then $K_1 \preceq^L K_2$.*

Remark 8.7 [Simulation of reachable states] If $K_1 \preceq^S K_2$ then for every reachable state s of K_1, there exists a reachable state t of K_2 such that t simulates s. ■

It follows that the language-inclusion problem (K_1, K_2) can be solved by exhibiting a simulation $\preceq$ of $K_1 + K_2$ such that for every initial state s of K_1 there is an initial state t of K_2 with $s \preceq t$.

8.3.1 Similarity

SIMILARITY

The state equivalence $\simeq^S$ induced by the simulation preorder $\preceq^S$ is called *similarity.*

Thus, $s \simeq^S t$ for two states s and t of the observation structure K if there exists a simulation $\preceq_1$ of K with $s \preceq_1 t$ and a simulation $\preceq_2$ of K with $t \preceq_2 s$. To observe that the similarity is more distinguishing than trace equivalence, consider states s and t of Figure 6.8 We have $s \simeq^L t$, but t does not simulate s.

Example 8.8 [Similarity vs. bisimilarity] To observe that the bisimilarity is more distinguishing than similarity, consider states s_0 and t_0 of Figure 8.5. The two states are not bisimilar. Observe that the relation

$$\{(s_0, t_0), (s_1, t_2), (s_2, t_4), (s_3, t_5)\}$$

is a simulation relation, and hence, t_0 simulates s_0. The relation

$$\{(t_0, s_0), (t_1, s_1), (t_2, s_1), (t_3, s_2), (t_4, s_2), (t_5, s_3)\}$$

is also a simulation relation, and hence, s_0 simulates t_0. Thus, the two states s_0 and t_0 are similar. ■

The relationship among various state equivalences is summarized in Theorem 8.6.

Theorem 8.6 [Distinguishing power of state equivalences] $\approx \;\sqsubset\; \simeq^L \;\sqsubset\; \simeq^S \;\sqsubset\; \simeq^B \;\sqsubset\; =$.

Exercise 8.10 {`T4`} [i-step similarity] (1) Define i-step trace equivalence and i-step similarity. Show that i-step similarity lies strictly between i-step trace equivalence and i-step bisimilarity on one hand, and between $(i-1)$-step similarity and $(i+1)$-step similarity on the other hand. (2) Give a fixpoint characterization of similarity. ■

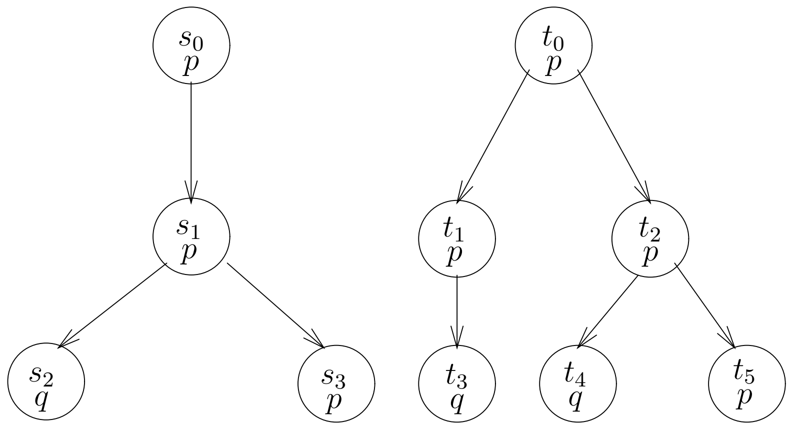

Figure 8.5: Comparing similarity and bisimilarity

Exercise 8.11 {T3} [Backward simulation] Let $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle \cdot \rangle\!\rangle)$ be an observation structure. A *backward simulation* $\preceq$ of K is a binary relation on the state space such that for all states s and t of K, if $s \preceq t$ then (1) $\langle\!\langle s \rangle\!\rangle = \langle\!\langle t \rangle\!\rangle$ and (2) if $s' \rightarrow s$, then there is a state t' such that $t' \rightarrow t$ and $s' \preceq t'$. The state t *backward simulates* the state s if there is a backward simulation $\preceq$ such that $s \preceq t$.

Prove that the language-inclusion problem (K_1, K_2) has the answer YES if there is a backward simulation $\preceq$ of $K_1 + K_2$ such that (1) for every state s of K_1 there is a state t of K_2 with $s \preceq t$, and (2) if s is initial and $s \preceq t$, then t is initial.

Prove that similarity and backward similarity are incomparable state equivalences. ■

Exercise 8.12 {T4} [Forward-backward simulation] (1) Prove that simulations and backward simulations are closed under relational composition, and show that the composition of a simulation with a backward simulation may be neither a simulation nor a backward simulation. (2) The composition of a simulation with a backward simulation is called a *forward-backward simulation*, and the composition of a backward simulation with a simulation is a *backward-forward simulation*. In this manner, we can define an infinite family of state equivalences. Prove that all members of this family lie strictly between trace equivalence and bisimilarity in distinguishing power. ■

8.3.2 Universal and existential STL

Universal and existential STL are the fragments of STL whose formulas do not contain quantifier switches. Since quantifier switches correspond to switching sides in the bisimilarity game, universal and existential STL cannot distinguish between similar states.

Let us recall the definition of *universal* STL (∀STL) from Chapter 6. The formulas of ∀STL are generated by the grammar

$$\phi \ ::= \ p \mid \phi \vee \phi \mid \phi \wedge \phi \mid \forall\bigcirc\, \phi \mid \phi \forall\mathcal{W}\phi.$$

Proposition 6.6 states that STL cannot distinguish between bisimilar states. Now, we establish that ∀STL cannot distinguish between similar states.

Proposition 8.6 [Simulation and Universal STL] *Let s and t be two states of an observation structure K, and let ϕ be a formula of* ∀STL. *Then, if $s \preceq^S_K t$ and $t \models \phi$ then $s \models \phi$.*

Exercise 8.13 {T2} [Simulation and Universal STL] Prove Proposition 8.6. ■

Corollary 8.5 [Similarity and Universal STL] *Similarity is an abstract semantics for* ∀STL*.*

It follows that it suffices to construct quotients with respect to similarity for model checking of ∀STL requirements. Since similarity is a coarser equivalence than bisimilarity, the quotient with respect to similarity can be smaller than the quotient with respect to bisimilarity.

Recall that bisimilarity is a fully abstract semantics for STL: the equivalence induced by STL coincides with bisimilarity. A similar result holds for ∀STL and similarity: two states of an observation structure K that are not similar can be distinguished by an ∀STL-formula.

Proposition 8.7 [∀STL full abstraction] *Similarity is a fully abstract semantics for* ∀STL*.*

Exercise 8.14 {T3} [Distinguishing non-similar states with ∀STL] Show that two non-similar states of a finitary observation structure can be distinguished by an ∀STL-formula that uses only the next-time operator $\forall\bigcirc$. Proposition 8.7 follows. ■

Exercise 8.15 {T3} [Existential STL] The formulas of *existential* STL (∃STL) are generated by the grammar

$$\phi \ ::= \ p \mid \phi \vee \phi \mid \phi \wedge \phi \mid \exists\bigcirc\, \phi \mid \phi \exists\mathcal{U}\phi.$$

(1) Let s and t be two states of an observation structure K, and let ϕ be a formula of ∃STL. Prove that if $s \preceq^S_K t$ and $s \models \phi$ then $t \models \phi$. It follows that two similar states satisfy the same ∃STL formulas, and similarity is an abstract semantics for ∃STL. (2) Let s and t be two non-similar states of an observation structure K. Prove that there exists an ∃STL-formula that is satisfied by only one of the two states. It follows that the equivalence induced by ∃STL coincides with similarity. ■

State Equivalence	Complexity	Logic
Trace equivalence $\simeq^L$	$O(m \cdot \mathbf{2}^n)$/PSPACE	SAL
Similarity $\simeq^S$	$O(m \cdot n)$	$\forall$STL, $\exists$STL
Bisimilarity $\simeq^B$	$O(m \cdot \log\ n)$	STL

Figure 8.6: Summary of state equivalences

8.4 Computing Similarity

We proceed to study algorithms for deciding whether one structure simulates another. As in case of partition refinement, both enumerative and symbolic algorithms are considered. The complexity of deciding the similarity relation on a finite observation structure is quadratic ($O(m \cdot n)$). Contrast this with $O(m \cdot \log\ n)$ complexity of deciding the bisimilarity relation, which is finer than similarity, and PSPACE complexity of deciding language equivalence, which is coarser than similarity.

The results concerning the three state equivalences, trace equivalence, similarity, and bisimilarity, are summarized in Figure 8.6. The second column shows complexity of deciding equivalence of two states in a structure with n states and m transitions, while the third column list the logic(s) for which the corresponding equivalence is fully abstract.

Let K be an observation structure, and let s be a state of K. Then, the *simulator set* $\mathit{sim}(s)$ of s is the set of states that simulate s.

An instance of the similarity-checking problem consists of a finite observation structure K. The answer to the similarity-checking problem is the set of simulator sets $\mathit{sim}(s)$, for each state s of K.

Remark 8.8 [Simulator sets] Let K be an observation structure. Similar states have identical simulator sets: for two states s and t of K, if $s \simeq^S t$ then $\mathit{sim}(s) = \mathit{sim}(t)$. The simulator set of every state is a block of the partition $\simeq^S$: for every state s, $\mathit{sim}(s)$ is a union of equivalence classes of $\simeq^S$. ■

Once the similarity-checking problem K is solved, then $s \simeq^S t$ iff $s \in \mathit{sim}(t)$ and $t \in \mathit{sim}(s)$. Similarity-checking problem can be used to decide if one observation structure simulates another.

8.4.1 Enumerative Similarity Checking

We develop our enumerative algorithm in three steps.

Algorithm 8.1 [Schematic Similarity]

Input: a finite observation structure $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle \cdot \rangle\!\rangle)$.
Output: for each state $s \in \Sigma$, the simulator set $\mathit{sim}(s)$.

```
foreach $s \in \Sigma$ do $\mathit{sim}(s) := \{t \in \Sigma \mid \langle\!\langle t \rangle\!\rangle = \langle\!\langle s \rangle\!\rangle\}$ od;
while there are three states $t$, $s$, and $u$ such that
        $s \in \mathit{post}(t)$, $u \in \mathit{sim}(t)$, and $\mathit{post}(u) \cap \mathit{sim}(s) = \emptyset$ do
  $\mathit{sim}(t) := \mathit{Delete}(u, \mathit{sim}(t))$
  {I0: assert for all $s, t \in \Sigma$, if $t$ simulates $s$ then $t \in \mathit{sim}(s)$}
  od.
```

Figure 8.7: Enumerative similarity checking

Schematic similarity

We start with the schematic algorithm shown in Figure 8.7. For each state s, the set $\mathit{sim}(s)$ contains states that are candidates for simulating s. Initially, $\mathit{sim}(s)$ contains all states with the observation of s. If $t \rightarrow s$ and $u \in \mathit{sim}(t)$, but there is no $v \in \mathit{sim}(s)$ such that $u \rightarrow v$, then u cannot simulate t and is removed from $\mathit{sim}(t)$, without violating the invariant assertion I0. In this case, we say that $\mathit{sim}(t)$ is *sharpened* with respect to the transition (t, s). It is easy to check that if no transitions allow a sharpening of $\mathit{sim}(t)$ for any state t, then for all s, all states in $\mathit{sim}(s)$ can simulate s.

Theorem 8.7 [Schematic similarity] *Given a finite observation structure K, Algorithm 8.1 correctly solves the similarity-checking problem.*

If the input structure has n states, there can be at most n^2 iterations of the while loop. A naive implementation of the schematic algorithm therefore requires time $O(m^2 n^3)$, where $m \geq n$ is the number of transitions of the input structure. We will improve the running time to $O(mn)$.

Refined similarity

The algorithm of Figure 8.8 refines the schema of Algorithm 8.1. The key idea of the refinement is the introduction of a set $\mathit{prevsim}(s)$ for each state s. For each state s, the set $\mathit{prevsim}(s)$ is a superset of $\mathit{sim}(s)$ and contains states that once were considered candidates for simulating s. The crucial invariant I2 of the while loop allows us to sharpen $\mathit{sim}(t)$ with respect to the transition (t, s) by looking only at states in $\mathit{prevsim}(s)$ when checking if a state $u \in \mathit{sim}(t)$ has a successor in $\mathit{sim}(s)$. Moreover, once $v \in \mathit{prevsim}(s) \backslash \mathit{sim}(s)$ is examined once, v is removed from $\mathit{prevsim}(s)$ forever.

Algorithm 8.2 [Refined Similarity]

Input: a finite observation structure $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$.
Output: for each state $s \in \Sigma$, the simulator set $\mathit{sim}(s)$.

```
foreach $s \in \Sigma$ do
  $\mathit{prevsim}(s) := \Sigma$;
  if $\mathit{post}(s) = \emptyset$
    then $\mathit{sim}(s) := \{t \in \Sigma \mid \langle\!\langle t\rangle\!\rangle = \langle\!\langle s\rangle\!\rangle\}$
    else $\mathit{sim}(s) := \{t \in \Sigma \mid \langle\!\langle t\rangle\!\rangle = \langle\!\langle s\rangle\!\rangle$ and $\mathit{post}(t) \neq \emptyset\}$
    fi
  od;
while there is a state $s \in \Sigma$ such that $\mathit{sim}(s) \neq \mathit{prevsim}(s)$ do
  {I1: assert for all $s \in \Sigma$, $\mathit{sim}(s) \subseteq \mathit{prevsim}(s)$}
  {I2: assert for all $t, s, u \in \Sigma$, if $t \rightarrow s$ and $u \in \mathit{sim}(t)$,
    then $\mathit{post}(u) \cap \mathit{prevsim}(s) \neq \emptyset$}
  $\mathit{remove} := \mathit{pre}(\mathit{prevsim}(s)) \backslash \mathit{pre}(\mathit{sim}(s))$;
  foreach $t \in \mathit{pre}(s)$ do $\mathit{sim}(t) := \mathit{sim}(t) \backslash \mathit{remove}$ od;
  $\mathit{prevsim}(s) := \mathit{sim}(s)$
  od.
```

Figure 8.8: Refined similarity checking

The initial **for** loop of Algorithm 8.2 performs, in addition to the work of the initial **for** loop of Algorithm 8.1, also some of the work of the **while** loop of Algorithm 8.1. For each state s, the set $\mathit{prevsim}(s)$ is initialized to contain all states, and $\mathit{sim}(s)$ is initialized to contain all states with the same observation as that of s, and that have a successor if s does. This initialization establishes the two invariants I1 and I2. In each iteration of the **while** loop, we nondeterministically pick a state s for which $\mathit{sim}(s)$ improves on $\mathit{prevsim}(s)$, and we sharpen $\mathit{sim}(t)$ for all predecessors t of s with respect to the transition (t, s). By I2, all states in $\mathit{sim}(t)$ have successors in $\mathit{prevsim}(s)$, and we can find all states in $\mathit{sim}(t)$ that do not have successors in $\mathit{sim}(s)$ by looking at the predecessor set of $\mathit{prevsim}(s)$. These states are collected in the set *remove* and deleted from $\mathit{sim}(t)$. Once all predecessors of s have been processed in this fashion, we update $\mathit{prevsim}(s)$ to $\mathit{sim}(s)$. If $\mathit{sim}(s) = \mathit{prevsim}(s)$ for all states s, then I2 implies the termination condition of the **while** loop of Algorithm 8.1.

Quadratic similarity checking

The algorithm of Figure 8.9 implements the scheme of Algorithm 8.2 using two data structures. First, instead of recomputing the set *remove* in each iteration of the **while** loop, the algorithm dynamically maintains for each state s a set $\mathit{remove}(s)$ that satisfies the invariant I3. If $\mathit{remove}(s) = \emptyset$ for all states s, then I1 and I3 imply the termination condition of the **while** loop of Algorithm 8.2. Second (not shown in the figure), we maintain a two-dimensional array $\mathit{count}[1..n, 1..n]$ of nonnegative integers such that $\mathit{count}[v, t] = |\mathit{post}(v) \cap \mathit{sim}(t)|$ for all states v and t. The array *count* is initialized in time $O(mn)$. Whenever a state u is removed from $\mathit{sim}(t)$, then the counters $\mathit{count}[v, t]$ are decremented for all predecessors v of u. The cost of these decrements is absorbed in the cost of the innermost **if** statement. With the array *count*, the test $\mathit{post}(v) \cap \mathit{sim}(t) = \emptyset$ of that if statement can be executed in constant time, by checking if $\mathit{count}[v, t] = 0$.

The initialization of $\mathit{sim}(s)$ for all s requires time $O(n \cdot (m + n))$. The initialization of $\mathit{remove}(s)$ for all s requires time $O(mn)$. Given two states s and u, if the test $u \in \mathit{remove}(s)$ is positive in iteration i of the **while** loop, then the test $u \in \mathit{remove}(s)$ is negative in all iterations $j > i$. This is because (1) in all iterations, $u \in \mathit{remove}(s)$ implies that $u \notin \mathit{pre}(\mathit{sim}(s))$, (2) the value of $\mathit{prevsim}(s)$ in all iterations $j > i$ is a subset of the value of $\mathit{sim}(s)$ in iteration i, and (3) invariant I1. It follows that the test $u \in \mathit{sim}(t)$ is executed $\Sigma_s \Sigma_u |\mathit{pre}(s)| = O(mn)$ times. The test $u \in \mathit{sim}(t)$ is positive at most once for every u and t, because after a positive test u is removed from $\mathit{sim}(t)$ and never put back. Therefore the body of the outer **if** statement in the **while** loop contributes time $\Sigma_u \Sigma_t (1 + |\mathit{pre}(u)|) = O(mn)$. This gives a total running time of $O(mn)$.

Algorithm 8.3 [Quadratic Similarity]

Input: a finite observation structure $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle \cdot \rangle\!\rangle)$.
Output: for each state $s \in \Sigma$, the simulator set $sim(s)$.

```
foreach $s \in \Sigma$ do
  {let $prevsim(s) := \Sigma$}
  if $post(s) = \emptyset$
    then $sim(s) := \{t \in \Sigma \mid \langle\!\langle t \rangle\!\rangle = \langle\!\langle s \rangle\!\rangle\}$
    else $sim(s) := \{t \in \Sigma \mid \langle\!\langle t \rangle\!\rangle = \langle\!\langle s \rangle\!\rangle$ and $post(t) \neq \emptyset\}$
    fi;
  $remove(s) := pre(\Sigma) \backslash pre(sim(s))$
  od;
while there is a state $s \in \Sigma$ such that $remove(s) \neq \emptyset$ do
  {I3: assert for all $s \in \Sigma$, $remove(s) = pre(prevsim(s)) \backslash pre(sim(s))$}
  foreach $t \in pre(s)$ do
    foreach $u \in remove(s)$ do
      if $u \in sim(t)$ then
        $sim(t) := Delete(u, Sim(t))$;
        foreach $v \in pre(u)$ do
          if $post(v) \cap sim(t) = \emptyset$ then $remove(t) := Insert(v, remove(t))$ fi
          od
        fi
      od
    od;
    {let $prevsim(s) := sim(s)$}
  $remove(s) := \emptyset$
  od.
```

Figure 8.9: Efficient similarity checking

Theorem 8.8 [Enumerative similarity checking] *Given a finite observation structure with n states and m transitions, Algorithm 8.3 solves the similarity checking problem in time $O(mn)$.*

Corollary 8.6 [Checking similarity of states] *The similarity of two states of a finite observation structure can be decided in time $O(mn)$.*

8.4.2 Symbolic Similarity Checking

Symbolic procedures operate on regions, rather than states. Instead of computing simulator sets for individual states, we compute simulator sets for entire regions. Recall that if two states are similar, then their simulator sets are identical, and the simulator set of every state is a block of the similarity relation $\simeq^S$. This suggests that we should compute simulator sets of equivalence classes of $\simeq^S$, rather than simulator sets of individual states. These two facts lead us to the following definition.

Symbolic simulator sets

Given an observation structure K, and a K-partition $\cong$, the simulator function for $\cong$ is the function *Sim* that maps each region σ in $\cong$ to the union $\bigcup_{s\in\sigma} \mathit{sim}(s)$. The *symbolic simulator structure* for K is the pair $(\simeq^S, \mathit{Sim})$ consisting of the similarity partition $\simeq^S$ and the simulator function *Sim* for $\simeq^S$.

Constructing the symbolic simulator structure suffices to answer the similarity checking problem: if $(\simeq^S, \mathit{Sim})$ is the symbolic simulator structure for K, then for a state s of K, $\mathit{sim}(s) = \mathit{Sim}(\sigma)$ where σ is the equivalence class of $\simeq^S$ that contains s.

For an observation a, let $\Sigma_a = \{s \in \Sigma \mid \langle\langle s\rangle\rangle = a\}$ be the region of states with the observation a. Thus, the collection $\{\Sigma_a \mid a \in A\}$ defines the partition induced by the propositional equivalence $\approx$. We develop our procedure in two steps.

Revised schematic similarity

We start with the schema shown in Figure 8.10, which relaxes the schema of Algorithm 8.1. The initial **for** loops are identical, and establish the two invariants I4 and I5. The invariant I5 asserts that whenever a simulator set $\mathit{sim}(s)$ contains a state u', and u'' simulates u', then $\mathit{sim}(s)$ contains also u''. Assuming I5, if $u \in \mathit{sim}(t)$, $u' \in \mathit{sim}(s)$, and $t \rightarrow u'$, but there is no $u'' \in \mathit{sim}(s)$ such that $u \rightarrow u''$, then u cannot simulate t. This is because in order for u to simulate t, some successor of u would have to simulate u', which is not possible, because by I5 all states that simulate u' are contained in $\mathit{sim}(s)$. We can therefore remove u from $\mathit{sim}(t)$, maintaining both invariants, even if Algorithm 8.1

Algorithm 8.4 [Revised Schematic Similarity]

Input: an observation structure $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle \cdot \rangle\!\rangle)$.
Output: for each state $s \in \Sigma$, the simulator set $sim(s)$.

```
foreach $s \in \Sigma$ do $sim(s)$ := $\{t \in \Sigma \mid \langle\!\langle t \rangle\!\rangle = \langle\!\langle s \rangle\!\rangle\}$ od;
while there are three states $s$, $t$, and $u$ such that
        $post(t) \cap sim(s) \neq \emptyset$, $u \in sim(t)$, and $post(u) \cap sim(s) = \emptyset$ do
    {I4: assert for all $s \in \Sigma$, $s \in sim(s)$}
    {I5: assert for all $s, t, u \in \Sigma$, if $t \preceq^S u$ and $t \in sim(s)$, then $u \in sim(s)$}
    $sim(t) := Delete(u, sim(t))$
    od.
```

Figure 8.10: Revised scheme for similarity checking

would not have allowed us to do so. In this case, we say that $sim(t)$ is *freely sharpened* with respect to the transition (t, u'). If the transition (t, s) allows a sharpening of $sim(t)$, then I4 implies that (t, s) also allows a free sharpening of $sim(t)$. Consequently, if no transitions allow a free sharpening of $sim(t)$ for any state t, then the termination condition of Algorithm 8.1 is satisfied. This implies the partial correctness of the revised scheme.

Theorem 8.9 [Revised schematic similarity] *Given a finite observation structure K, Algorithm 8.4 correctly solves the similarity-checking problem.*

Symbolic algorithm

The symbolic procedure, shown in Figure 8.11, is an instance of the schema of Algorithm 8.4. The symbolic algorithm uses a symbolic representation of regions. The only primitive operations it needs are boolean operations and the *pre*-operation on regions, and emptiness checking of regions. Thus, it is not restricted to finite observation structures, but rather to those structures that support an effective symbolic representation of regions. If the similarity relation $\simeq^S$ of the input structure is finite, then it has only finitely many blocks, and the invariant I7 ensures that Algorithm 8.5 terminates. If $\simeq^S$ is infinite, then the partition $\cong$ needs to be refined infinitely often, and the procedure does not terminate.

In implementing Algorithm 8.5, we can enforce the invariant that for all regions $\sigma \in \cong$, the region $Sim(\sigma)$ is a block of $\cong$, by refining the partition $\cong$ whenever this becomes necessary due to the creation of a new simulator set. Such an implementation maintains a finite partition $\cong$ of the state space Σ together

Algorithm 8.5 [Symbolic Similarity]

Input: an observation structure $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$.
Output: the symbolic simulator structure $(\simeq^S, \mathit{Sim})$ of K.

```
$\cong$ := $\{\Sigma_a \mid a \in A$ and $\Sigma_a \neq \emptyset\}$;
foreach $\sigma \in \cong$ do $\mathit{Sim}(\sigma) := \sigma$ od;
while there are two regions $\sigma, \tau \in \cong$ such that $\sigma \cap \mathit{pre}(\mathit{Sim}(\tau)) \neq \emptyset$
        and $\mathit{Sim}(\sigma) \backslash \mathit{pre}(\mathit{Sim}(\tau)) \neq \emptyset$ do
    {I6: assert for all $\sigma \in \cong$ and all $s \in \sigma$, $\mathit{sim}(s) = \mathit{Sim}(\sigma)$}
    {I7: assert for all $\sigma \in \cong$, both $\sigma$ and $\mathit{Sim}(\sigma)$ are blocks of $\simeq^S$}
    $\sigma' := \sigma \cap \mathit{pre}(\mathit{Sim}(\tau))$;  $\sigma'' := \sigma \backslash \mathit{pre}(\mathit{Sim}(\tau))$;
    $\cong := \mathit{Insert}(\sigma', \mathit{Delete}(\sigma, \cong))$;
    $\mathit{Sim}(\sigma') := \mathit{Sim}(\sigma) \cap \mathit{pre}(\mathit{Sim}(\tau))$;
    if $\sigma'' \neq \emptyset$ then $\cong := \mathit{Insert}(\sigma'', \cong)$; $\mathit{Sim}(\sigma'') := \mathit{Sim}(\sigma)$ fi
    od.
```

Figure 8.11: Symbolic similarity checking

with pointers from each region σ in $\cong$ to all regions υ in $\cong$ with $\upsilon \subseteq \mathit{Sim}(\sigma)$, without representing the simulator set $\mathit{Sim}(\sigma)$ explicitly.

Theorem 8.10 [Symbolic similarity checking] *Given an observation structure K with a finite similarity relation, Algorithm 8.5 terminates and computes the symbolic simulator structure for K.*

8.5 Hierarchical Reasoning

Establishing that a reactive module P implements another module Q is computationally hard. In this section, we consider simulation relations as a sufficient condition for establishing the implementation relation between two modules.

8.5.1 Simulation preorder over modules

Every state preorder for observation structures leads to a preorder over modules. Thus, the simulation preorder $\preceq^S$ can be used to compare one module with another. The reactive module Q *simulates* the reactive module P, denoted $P \preceq^S Q$, if (1) every interface variable of Q is an interface variable of P: $\mathsf{intf}X_Q \subseteq \mathsf{intf}X_P$, (2) every external variable of Q is an observable variable of P: $\mathsf{extl}X_Q \subseteq \mathsf{obs}X_P$, (3) for all variables x in $\mathsf{obs}X_Q$ and y in $\mathsf{intf}X_Q$, if $y \prec_Q x$ then $y \prec_P x$, and (4) the observation structure K_Q simulates the observation structure $K_{P'}$ for $P' = \textbf{hide}\ (\mathsf{obs}X_P \backslash \mathsf{obs}X_Q)\ \textbf{in}\ P$.

As in case of the implementation preorder, if $P \preceq^S Q$, then the module P is more constrained than P. The fourth requirement can informally be read as "whatever P does is allowed by Q." Again, we can think of Q as the (more abstract) specification, and P as the (more detailed) implementation. The relationship between simulation preorder and language preorder over states leads to:

Proposition 8.8 [Simulation and implementation] *For two reactive modules P and Q, if $P \preceq^S Q$ then $P \preceq^L Q$.*

Remark 8.9 [Simulation and $\forall$STL] Suppose $P \preceq^S Q$, and let ϕ be a formula of $\forall$STL. If the answer to the verification problem (Q, ϕ) is YES, then the answer to the verification problem (P, ϕ) is also YES. ■

Given two modules P and Q such that $\mathsf{obs}X_Q \subseteq \mathsf{obs}X_P$, a state t of the module P simulates a state s of P, if $s \preceq^S t$ in the observation structure $K_{P'} + K_Q$ for $P' = \textbf{hide}\ (\mathsf{obs}X_P \backslash \mathsf{obs}X_Q)\ \textbf{in}\ P$. If $P \preceq^S Q$ then every reachable state of P is simulated by some reachable state of Q.

Example 8.9 [Nondeterministic versus deterministic scheduling] Recall the modules *Scheduler* and *NonDetScheduler* from Example 8.2. For a state s of *Scheduler* and a state t of *NonDetScheduler*, let $s \preceq t$ if the two states assign the same values to the variables $task_1$, $task_2$, $proc$, new_1, and new_2. Verify that $\preceq$ is a simulation relation: every transition of *Scheduler* is allowed by *NonDetScheduler*. ■

Example 8.10 [Synchronous versus asynchronous mutual exclusion] Let us revisit the two solutions to the mutual exclusion problem, namely, the modules *SyncMutex* and *Pete*. In Example 8.1, we established that *SyncMutex* $\preceq^L$ *Pete*. However, *SyncMutex* $\preceq^S$ *Pete* does not hold. To see this, first note that if a state t of *Pete* simulates a state s of *SyncMutex* then $pc_1[s] = pc_1[t]$ and $pc_2[s] = pc_2[t]$. Consider the following trajectory of *SyncMutex*:

$$\begin{array}{llll} s_0 : (outC, outC) & \to & s_1 : (outC, reqC) & \to \\ s_2 : (outC, inC) & \to & s_0 : (outC, outC) & \to \\ s_3 : (reqC, reqC) & \to & s_4 : (inC, reqC) & \end{array}$$

If $s_3 \preceq^S t_3$, then $x_1[t_3] \neq x_2[t_3]$. This is because in *Pete* if both processes are requesting and $x_1 = x_2$, then P_2 enters the critical section first, and hence, cannot match the transition from s_3 to s_4. This implies that if $s_0 \preceq^S t_0$, then $x_1[t_0] = x_2[t_0]$ (since t_0 is required to be a predecessor of t_3). Continuing this line of reasoning, if $s_2 \preceq^S t_2$, then $x_1[t_2] = x_2[t_2]$; if $s_1 \preceq^S t_1$, then $x_1[t_1] = x_2[t_1]$. Now there is no transition between t_0 and t_1, and thus, no such simulation exists. ■

Exercise 8.16 {`P2`} [Synchronous vs. asynchronous message passing] Recall the modules *SyncMsg* and *AsyncMsg* for synchronous and asynchronous message passing protocols. Does $\mathit{SyncMsg} \preceq^S \mathit{AsyncMsg}$ hold? ■

8.5.2 Compositional reasoning

Consider the implementation problem of verifying that a module implements its specification. As explained in Section 8.2, this task can be decomposed into subtasks using the compositional and modular properties of the implementation preorder. To verify a particular subtask $P \preceq^L Q$, we can try to prove the stronger goal $P \preceq^S Q$. To establish $P \preceq^S Q$, we can use the symbolic algorithms for similarity checking.

It turns out that the simulation preorder itself is compositional. Thus, if $P \preceq^S Q$ then $P \| R \preceq^S Q \| R$. This helps in decomposing the verification problem for $\forall$STL: if $P \preceq^S Q$ then all $\forall$STL formulas satisfied by $Q \| R$ are also satisfied by $P \| R$.

Proposition 8.9 [Compositionality of simulation] *The simulation preorder* $\preceq^S$ *on modules is compositional.*

Proof. Consider two reactive modules P and Q such that $P \preceq^S Q$. The cases corresponding to the operations of hiding and renaming are straightforward. We consider only parallel composition. Let R be a reactive module that is compatible with P. For a state s of $P \| R$ and a state t of $Q \| R$, let $s \preceq t$ iff (1) $X_R[s] = X_R[t]$, and (2) $X_Q[t]$ simulates $X_P[s]$.

We first show that $\preceq$ is a simulation relation. Consider $s \preceq t$ and s' be a successor of s in $P \| R$. Then, $X_Q[t]$ simulates $X_P[s]$, and $X_P[s']$ is a successor of $X_P[s]$. Since $P \preceq^S Q$, there exists a state t' of Q such that t' is a successor of $X_Q[t]$ in Q, and t' simulates $X_P[t]$. Let t'' be the state of $Q \| R$ such that $X_Q[t''] = t'$ and $X_R[t''] = X_R[t]$. By definition, $s' \preceq t''$. Since $X_R[s']$ is a successor of $X_R[s]$ in R, it follows that $X_R[t'']$ is a successor of $X_R[t]$ in R. Thus, t'' is a successor of t in $Q \| R$.

Along the same lines, we can establish that for every initial state s of $P \| R$, there is an initial state t of $Q \| R$ such that $s \preceq t$. ■

Exercise 8.17 {`T5`} [Assume-Guarantee for Simulation] Does the assume-guarantee theorem for implementation preorder (Theorem 8.4) hold for the simulation preorder $\preceq^S$? ■

8.5.3 Refinement mappings

Refinement maps, or homomorphisms, are special types of simulation relations.

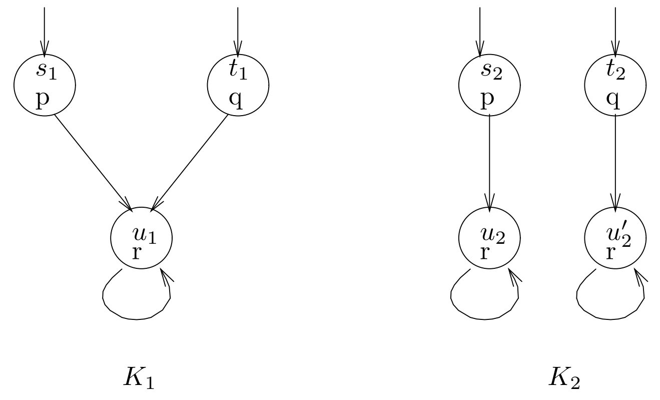

Figure 8.12: Refinement maps versus simulation relations

Refinement maps

Let $K_1 = (\Sigma_1, \sigma_1^I, \rightarrow_1, A, \langle\!\langle\cdot\rangle\!\rangle_1)$ and $K_2 = (\Sigma_2, \sigma_2^I, \rightarrow_2, A, \langle\!\langle\cdot\rangle\!\rangle_2)$ be two observation structures. A *refinement mapping* hom from K_1 to K_2 is a function from the reachable region σ_1^R of K_1 to Σ_2 such that (1) for all $s \in \sigma_1^R$, $\langle\!\langle \mathrm{hom}(s)\rangle\!\rangle_2 = \langle\!\langle s\rangle\!\rangle_1$, (2) for every reachable transition $s \rightarrow_1 t$ of K_1, $\mathrm{hom}(s) \rightarrow_2 \mathrm{hom}(t)$, and (3) for all $s \in \sigma_1^I$, $\mathrm{hom}(s) \in \sigma_2^I$.

If hom is a refinement map from K_1 to K_2 then the set $\{(s, \mathrm{hom}(s)) \mid s \in \sigma_1^R\}$ is simulation relation over the union $K_1 + K_2$.

Proposition 8.10 [Refinement maps and simulations] *If there exists a refinement mapping from the observation structure K_1 to K_2 then $K_1 \preceq^S K_2$.*

For two observation structures K_1 and K_2, if there exists a refinement map from K_1 to K_2, then $K_1 \preceq^S K_2$, and hence, $K_1 \preceq^L K_2$. Thus, we can establish implementation relation between two modules by supplying a refinement map from the states of the detailed module to the states of the abstract module. Existence of simulation relation between two modules, however, does not guarantee existence of refinement maps.

Example 8.11 [Nondeterministic versus deterministic scheduling] Recall the modules *Scheduler* and *NonDetScheduler* from Example 8.9. Given a state s of *Scheduler*, let hom(s) be the state of *NonDetScheduler* obtained by simply discarding the value of the variable *prior*. In this case, this projection map is a refinement map, and establishes that *Scheduler* $\preceq^S$ *NonDetScheduler*. ■

Example 8.12 [Refinement map vs. simulation] Consider the two observation structures K_1 and K_2 shown in Figure 8.12. The relation $\{(s_1, s_2), (t_1, t_2), (u_1, u_2), (u_1, u'_2)\}$

is a simulation relation, and thus, $K_1 \preceq^S K_2$. However, there is no refinement map from K_1 to K_2. Observe that there is a refinement map from K_2 to K_1, and $K_2 \preceq^S K_1$. ■

Exercise 8.18 {`T4`} [Completeness of refinement mappings] Let P and Q be two reactive modules such that $P \preceq^L Q$. Prove that there is a monitor R for P such that there is a refinement mapping from $K_{P\|R}$ to K_Q. ■

Exercise 8.19 {`P3`} [Verifying refinements] Write an algorithm that given two observation structures K_1 and K_2 and a mapping hom from the states of K_1 to the states of K_2 verifies whether or not hom is a refinement map. ■

8.6 Stutter-closed Implementation

In Chapter 6, we saw how each state equivalence leads to its stutter-closed version obtained by adding extra transitions that obliterate the distinction due to the number of rounds for which an observation stays unchanged. In the same manner, every state preorder leads to a stutter-closed version: two states of an observation structure K are related according to the stutter-closed version of a preorder $\preceq$, if those two states are related according to $\preceq$ in the stutter-closure of K.

Stutter closure of state preorders

Let $\preceq$ be a state preorder, and let K be an observation structure. For two states s and t of K, $s \underline{\preceq}_K t$, for the stutter closure $\underline{\preceq}$ of $\preceq$, if $s \preceq_{K^S} t$. The induced state preorder $\underline{\preceq}$ is called the *stutter closure* of $\preceq$.

Remark 8.10 [Alternative characterization of stutter closure of trace preorder] Let A be a set of symbols. Let $\overline{a}_{0\ldots m}$ be a word over A. A *stutter-extension* of $\overline{a}$ is a word that can be obtained from $\overline{a}$ by repeating each symbol of $\overline{a}$ finitely many times: a word $\overline{b}$ over A is a *stutter-extension* of $\overline{a}$ iff there exist positive integers $i_0, i_1, \ldots i_m$ such that $\overline{b} = a_0^{i_0} a_1^{i_1} \ldots a_m^{i_m}$. For two states s and t of an observation structure K, $s \underline{\preceq}^L t$ holds if for every source-s trace $\overline{a}$ there exists a source-t trace $\overline{b}$ and a word $\overline{c}$ such that $\overline{c}$ is a stutter-extension of $\overline{a}$ and is also a stutter-extension of $\overline{b}$. ■

Stutter closure of trace equivalence is the weakest equivalence we have considered so far: it is less distinguishing than trace equivalence, and it is less distinguishing than weak bisimilarity.

The stutter closure of the trace preorder over observation structures leads to a way of comparing two modules, called *weak implementation*, denoted $\underline{\preceq}^L$.

Example 8.13 [Equivalence of synchronous vs.asynchronous message passing] Recall the modules *SyncMsg* and *AsyncMsg* for synchronous and asynchronous message passing protocols. The two modules are not trace equivalent, however, they are equivalent according to the equivalence induced by weak implementation. ■

The weak-implementation relation plays an important role in reasoning about asynchronous systems.

Remark 8.11 [Stutter-extensions and asynchronous modules] If P is an asynchronous module, then its language L_P is closed under stutter-extension: if $\overline{a}$ is a trace of P then every stutter-extension of $\overline{a}$ is also a trace of P. ■

The weak implementation relation is compositional as long as we use only asynchronous modules.

Theorem 8.11 [Compositionality of weak implementation] *For two modules P and Q, if $P \underline{\preceq}^L Q$ then (1) for variable x of P,* **hide** x **in** $P \underline{\preceq}^L$ **hide** x **in** Q*; (2) for every variable renaming ρ, $P[\rho] \underline{\preceq}^L Q[\rho]$. For asynchronous modules P, Q, and R, if $P \underline{\preceq}^L Q$ and R is compatible with P, then R is compatible with Q and $P \| R \underline{\preceq}^L Q \| R$.*

Exercise 8.20 {T3} [Compositionality of weak implementation] Prove Theorem 8.11. Show that the preorder $\underline{\preceq}^L$ is not compositional with respect to parallel composition with all modules; that is, find modules P, Q, and R such that $P \underline{\preceq}^L Q$, but $P \| R \underline{\preceq}^L Q \| R$ does not hold. ■

The assume-guarantee theorem for the implementation relation holds for the weak-implementation relation provided we consider only asynchronous modules.

Theorem 8.12 [Assume-guarantuee reasoning for weak implementation] *Let P_1 and P_2 be two compatible asynchronous reactive modules, and let Q_1 and Q_2 be two compatible asynchronous reactive modules. If $P_1 \| Q_2 \underline{\preceq}^L Q_1$, $Q_1 \| P_2 \underline{\preceq}^L Q_2$, and every external variable of $Q_1 \| Q_2$ is an observable variable of $P_1 \| P_2$, then $P_1 \| P_2 \underline{\preceq}^L Q_1 \| Q_2$.*

Exercise 8.21 {T3} [Assume-guarantuee for weak implementation] Prove Theorem 8.12. ■

Weak similarity

The stutter closure $\underline{\preceq}^S$ of simulation preorder is called *weak simulation*, and the stutter closure $\cong^S$ of similarity is called *weak similarity.*

$$\begin{tikzcd}
\text{Weak Trace Equivalence } \cong^L \arrow[r, phantom, "\sqsubset"] \arrow[d, phantom, "\sqcap"] & \text{Trace Equivalence } \simeq^L \arrow[d, phantom, "\sqcap"] \\
\text{Weak Similarity } \cong^S \arrow[r, phantom, "\sqsubset"] \arrow[d, phantom, "\sqcap"] & \text{Similarity } \simeq^S \arrow[d, phantom, "\sqcap"] \\
\text{Weak Bisimilarity } \cong^B \arrow[r, phantom, "\sqsubset"] & \text{Bisimilarity } \simeq^B
\end{tikzcd}$$

Figure 8.13: Relationship among state equivalences

Remark 8.12 [Weak simulation] Weak simulation (and weak similarity) can be defined directly without considering the stutter closure operation on structures explicitly. Let $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$ be an observation structure. A *weak-simulation* $\underline{\preceq} \subseteq \Sigma^2$ of K is a binary relation on the state space such that for all states s and t of K, if $s \,\underline{\preceq}\, t$ then (1) $\langle\!\langle s\rangle\!\rangle = \langle\!\langle t\rangle\!\rangle$ and (2) if $s \rightarrow s'$, then there is a state t' such that $s' \,\underline{\preceq}\, t'$ and there exists a trajectory $\overline{t}_{0\ldots m}$ of K with $t_0 = t$, $t_m = t'$, and $\langle\!\langle t_i\rangle\!\rangle = \langle\!\langle t\rangle\!\rangle$ for $0 \leq i < m$. The state t *weakly-simulates* the state s if there is a weak simulation $\underline{\preceq}$ such that $s \,\underline{\preceq}\, t$. Now, $s \,\underline{\preceq}^S\, t$ if t weakly-simulates s. ■

We know that similarity is more distinguishing than trace equivalence, but less distinguishing than bisimilarity. Analogously, weak similarity is more distinguishing than weak trace equivalence, but less distinguishing than weak bisimilarity.

Exercise 8.22 {T3} [Weak similarity vs. trace equivalence] Establish that weak similarity and trace equivalence are incomparable. ■

Exercise 8.23 {T4} [Weak similarity and waiting-for fragment of $\forall$Stl] Let $\forall\text{Stl}^{\mathcal{W}}$ be the fragment of $\forall$Stl that contains no next-time operators, that is, its formulas are generated by the grammar

$$\phi \ ::= \ p \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \forall \mathcal{W} \phi.$$

Establish that weak similarity is a fully abstract semantics for this fragment. ■