Contents

Chapter 7

Automata-theoretic Safety Verification

Automata provide an alternative to temporal logic for specifying requirements of reactive modules. In the automata-theoretic approach to verification, a module is viewed as a generator of a formal language over the set of observations. The requirement is specified by an automaton that accepts only the desired behaviors. The verification problem, then, reduces to a language-inclusion problem: whether every sequence of observations generated by the module is accepted by the requirements automaton.

7.1 Automata

Languages over observations

The execution of a transition graph G results in a trajectory of G, which is a finite sequence of states. The execution of an observation structure K results in a trajectory of the underlying transition graph, and each state of the trajectory has an associated observation. The resulting finite sequence of observations is called a *trace* of K. The set of traces corresponding to initialized trajectories is a language over the set A of observations, and is the language generated by K.

TRACES

Let $K = (G, A, \langle\langle\cdot\rangle\rangle)$ be an observation structure. A *trace* of K is a nonempty word $\overline{a}$ over the alphabet A of observations such that there is a trajectory $\overline{s}$ of G with $\overline{a} = \langle\langle\overline{s}\rangle\rangle$. The word $\overline{a}$ is a *source-s trace* of K, for a state s of K, if there is a source-s trajectory $\overline{s}$ of G with $\overline{a} = \langle\langle\overline{s}\rangle\rangle$. The word $\overline{a}$ is an *initialized trace* of K if there is an initialized trajectory $\overline{s}$ of G with $\overline{a} = \langle\langle\overline{s}\rangle\rangle$. We write $L_K(s)$ for the set of source-s traces of K, and L_K for the set of initialized traces of K. The set L_K is called the *language* of K. For a module P, the set L_{K_P} is called the language of P.

Example 7.1 [Traces] A possible initialized trace of the observation structure of Figure 6.1 is the word *pqprrrpr*. The language of initialized traces is a regular language specified by the expression $p + (pq((\epsilon + p)r)^*)^+(\epsilon + p)$.

The language of the module *Pete* of Chapter 1 is a regular language over the alphabet $\{outC, reqC, inC\} \times \{outC, reqC, inC\}$. A possible initialized trace of the module *Pete* is the word

$$\begin{array}{llll} (outC, outC), & (outC, reqC), & (reqC, inC), & (reqC, inC), \\ (reqC, outC), & (reqC, reqC), & (inC, reqC), & (outC, reqC), \\ (outC, inC), & (reqC, outC), & (inC, outC), & (outC, outC) \end{array}$$

corresponding to the initialized trajectory of Figure 2.1. ■

Remark 7.1 [Closure properties of trace languages] For every observation structure K, the language L_K is prefix-closed, but not necessarily fusion-closed. For example, for the observation structure shown in Figure 6.1, both *pqrpq* and *pqp* are initialized traces, but *pqpq* is not an initialized trace. This simply says that an observation may be caused by many different states and, by itself, does not necessarily determine the future behavior of the structure. ■

Exercise 7.1 {T2} [Prefix-closure] If K is an observation structure with the observation alphabet A, then the set L_K of initialized traces is a prefix-closed language over the alphabet A. Conversely, let A be an alphabet, and let L be a prefix-closed language over A. Show that there is an observation structure K such that $L = L_K$. ■

From observation structures to automata

Since languages of observation structures are prefix-closed, observation structures are not closed under complementation. To define languages that are not prefix-closed, we add acceptance conditions to observation structures.

AUTOMATON

An *automaton* M consists of (1) an observation structure K and (2) [the *accepting region*] a region σ^A of K. An initialized trajectory $\overline{s}_{0..m}$ of K is *accepted* by the automaton M if $s_m \in \sigma^A$. An initialized trace $\overline{a}$ of K is *accepted* by M if $\overline{a} = \langle\langle \overline{s} \rangle\rangle$ for some initialized trajectory $\overline{s}$ of K that is accepted by M. The *language* L_M of the automaton M is the set of initialized accepted traces of M.

The language of an automaton is a subset of the language of the underlying observation structure. Let K be an observation structure with the state space Σ. Declaring every state of K as accepting, we obtain the automaton $M_K = (K, \Sigma)$. For every observation structure K, $L_{M_K} = L_K$. Thus, every observation structure can be considered as an automaton, and sometimes we will not make the distinction between the observation structure K and the automaton M_K.

Remark 7.2 [Automaton definition] Our definition of an automaton is similar to the common definitions found in the textbooks on formal languages. In particular, if the automaton M has finitely many states, then the language L_M is regular. In the more common definition of an automaton, the transitions of the automaton are labeled with alphabet symbols. This is dual to our definition in which the states are labeled with observations. As a consequence of our definition, the empty word ϵ does not belong to the language of any automaton. For every regular language L over a finite alphabet A, there exists a finite automaton M such that $L_M = L \backslash \{\epsilon\}$. ■

The language-inclusion problem

The language-inclusion problem asks whether every initialized accepted trace of one automaton is also an initialized accepted trace of another automaton.

THE LANGUAGE-INCLUSION PROBLEM

An instance (M_1, M_2) of the *language-inclusion problem* consists of two automata M_1 and M_2 over the same observation alphabet A. The answer to the language-inclusion problem (M_1, M_2) is YES if $L_{M_1} \subseteq L_{M_2}$, and otherwise NO.

7.2 Safe Automaton Logic

7.2.1 Syntax and Semantics

Automata can be used for specifying requirements of reactive modules. The observations of the requirements automaton are boolean expressions over the observable variables of modules. We define the state logic SAL whose formulas are boolean combinations of such automata.

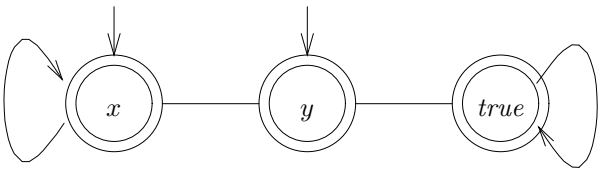


Figure 7.1: The automaton $M_{\mathcal{W}}$

SAFE AUTOMATON LOGIC

The formulas of the state logic *Safe automaton logic* (SAL) are generated by the grammar

$$\phi \ ::= \ \forall M \mid \neg\phi \mid \phi \vee \phi$$

where M is an automaton whose observations are boolean expressions.

Given a formula ϕ of SAL, an observation structure K is a ϕ-structure if each observation of K is a valuation for a superset of the variables appearing in the observations of all automata occurring in ϕ.

The satisfaction relation for SAL is defined by:

$s \models_K \forall M$	iff	for every source-s trajectory $\overline{s}_{0..m}$ of K there is a trace $\overline{a}_{0..m} \in L_M$ such that for all $0 \leq i \leq m$, $s_i \models a_i$
$s \models_K \phi \vee \psi$	iff	$s \models_K \phi$ or $s \models_K \psi$;
$s \models_K \neg\phi$	iff	$s \not\models_K \phi$,

where M is an automaton, ϕ and ψ are SAL formulas, and K is a (M, ϕ, ψ)-structure.

In other words, a state s of K satisfies the requirement specified by the formula $\forall M$ if for every source-s trace $\overline{a}$ of K, we can find an initialized accepting trace $\overline{b}$ of M such that every observation in $\overline{a}$ is consistent with the corresponding expression in $\overline{b}$. The characteristic region of an SAL formula, the satisfaction of an SAL formula by an observation structure, the model-checking problem for SAL, and the verification problem for SAL are defined as in case of other state logics such as STL.

Example 7.2 [Specifying $x\forall\mathcal{W}y$ in SAL] The SAL formula $\forall M_{\mathcal{W}}$ for the automaton shown in Figure 7.1 asserts that, given a state s, along every source-s trajectory a state that violates x coincides with or is preceded by a state that satisfies y. The formula $\forall M_{\mathcal{W}}$ can be interpreted at states of an observation structure whose observations assign values to x and y. It follows that the SAL formula $\forall M_{\mathcal{W}}$ is equivalent to the STL formula $x\forall\mathcal{W}y$. ■

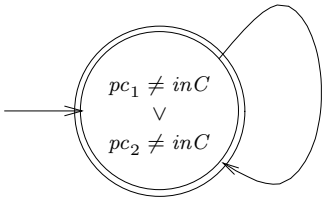
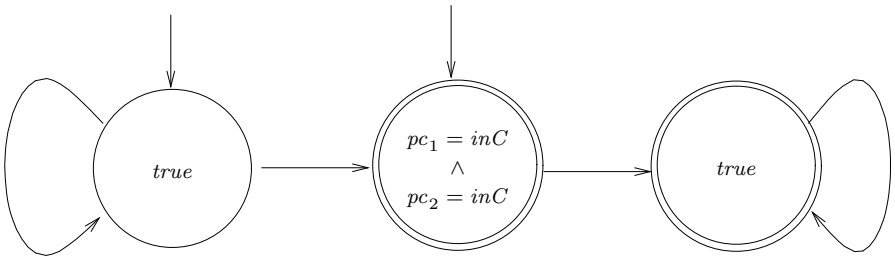


Figure 7.2: The mutual-exclusion requirement in SAL



Automaton $M_{\neg mutex}$

Figure 7.3: The violation of mutual-exclusion requirement in SAL

Example 7.3 [Mutual exclusion] Recall mutual-exclusion protocols from Chapter 1. The mutual-exclusion requirement can be expressed in SAL by the automaton M_{mutex} shown in Figure 7.2. That is, the SAL formula $\forall M_{mutex}$ is equivalent to the STL formula $\forall\Box(pc_1 \neq inC \vee pc_2 \neq inC)$. Notice that allowing the predicate $(pc_1 \neq inC \vee pc_2 \neq inC)$ as an observation in the automaton M_{mutex} permits a compact description of the property. ■

Remark 7.3 [Final states] Notice that both automata $M_{\mathcal{W}}$ and M_{mutex} are really observation structures, because all their states are accepting. This is no accident. Given an automaton M, let M^+ be the automaton that accepts the maximal prefix-closed subset of L_M; that is, $\overline{a} \in L_{M^+}$ if all prefixes of $\overline{a}$ are in L_M (how would you construct M^+?). Clearly, L_{M^+} can be accepted by an automaton all of whose states are accepting (remove the states that cannot reach an accepting state and make all remaining states accepting). If we specify a requirement of a state s of structure K by the SAL formula $\forall M$, then $s \models_K M$ iff $s \models_K M^+$. So we may specify the same requirement as $\forall M^+$.

We use automata (with accepting states) for specifying existential properties. Given an automaton M, let $-M$ be the complementary automaton (complementation requires accepting states; see below). In state s, the SAL formula $\neg\forall - M$ specifies that *some* trace from s is a trace of M. Therefore, we define

$\exists M$ as $\neg\forall - M$. For example, mutual exclusion can be specified by the SAL formula $\neg\exists M_{\neg mutex}$, where the automaton $M_{\neg mutex}$ of Figure 7.3 accepts all traces that cause violation of the mutual exclusion requirement. ■

Exercise 7.2 {T2} [Mutual exclusion] Express in SAL the first-request-first-in requirement and the equal-opportunity requirement for mutual-exclusion protocols (see Chapter 6 for the requirements). ■

Exercise 7.3 {T2} [Specifying $\forall\Box(x\forall\mathcal{W}y)$ in SAL] The automaton of Figure 7.1 expresses a requirement that is equivalent to the STL formula $x\,\forall\mathcal{W}\,y$. Write an automaton M such that $\forall M$ is equivalent to the STL formula $\forall\Box(x\,\forall\mathcal{W}\,y)$. ■

SAL model checking

The model-checking problem for SAL can be reduced to the language-inclusion problem. First, since every SAL formula is a boolean combination of automata, it is sufficient to consider the problem of checking whether an observation structure satisfies an automaton specification. For this purpose, we expand each automaton M of SAL to a larger automaton EM whose observations are valuations to the variables appearing in the observations of M.

> EXPANSION OF A SAL AUTOMATON
>
> For a SAL automaton M, its *expansion* EM is another automaton with the following components. (1) [Observation alphabet] Observations of EM are the valuations for the variables appearing in the observations of M. (2) [State space] For every state s of M, EM has a state $\langle s,t\rangle$ for each observation t of EM such that t satisfies the observation of s. (3) [Initial region] The state $\langle s,t\rangle$ is initial in EM if the state s is initial in M. (4) [Transition relation] There is a transition from $\langle s,t\rangle$ to $\langle s',t'\rangle$ in EM if there is a transition from s to s' in M. (5) [Observation function] The observation of $\langle s,t\rangle$ is t. (6) [Accepting region] The state $\langle s,t\rangle$ is accepting in EM if the state s is accepting in M.

Example 7.4 [SAL model checking] Figure 7.4 shows the observation structure $EM_{\mathcal{W}}$ for the SAL automaton $M_{\mathcal{W}}$ from Figure 7.1 (all states are accepting). ■

Thus, the observations of the automaton EM completely specify the values of the variables mentioned in the expressions labeling the original automaton M. Consider a M-structure K, and a trajectory $\overline{s}_{0\ldots m}$ of K. There exists a (initialized accepting) trace $\overline{a}_{0\ldots m}$ of M such that $\langle\langle s_i\rangle\rangle \models a_i$ for $0 \le i \le m$, iff $\langle\langle\overline{s}\rangle\rangle$ is an (initialized accepting) trace of EM. It follows that checking whether an observation structure satisfies an SAL automaton M is equivalent to checking whether K satsifies the expanded automaton EM, which in turn corresponds to checking whether the language of K is contained in the language of EM.

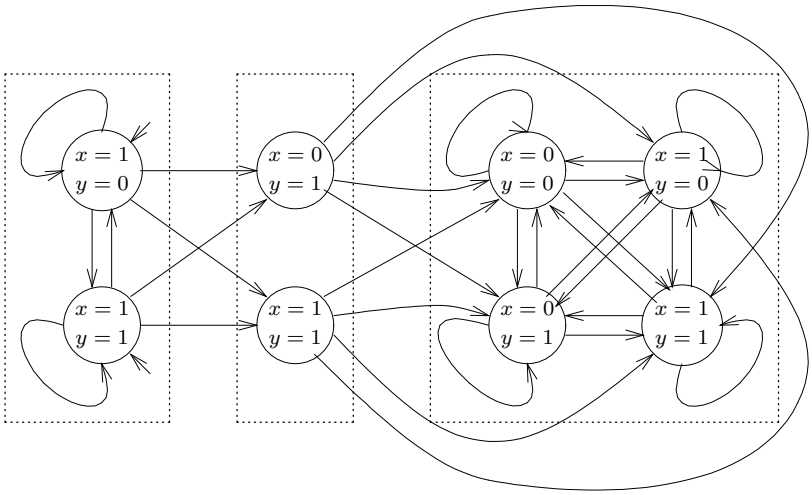


Figure 7.4: The automaton $EM_{\mathcal{W}}$

Proposition 7.1 [SAL model checking] *The* SAL *model-checking problem* (K, M) *and the language-inclusion problem* (K, EM) *have the same answer.*

Exercise 7.4 {T3} [SAL verification vs. invariant-verification with monitors] Can a given SAL-verification problem always be reduced to an invariant-verification problem after introducing a monitor (see Chapter 2 for monitors)? What about the converse? ■

7.2.2 The Distinguishing Power of SAL

The set of traces associated with a state of an observation structure leads to a natural way of equating two states: two states s and t of an observation structure are *trace equivalent* iff every source-s trace is also a source-t trace, and vice versa.

TRACE EQUIVALENCE

Two states s and t of an observation structure K are *trace equivalent*, denoted $s \simeq^L_K t$, if $L_K(s) = L_K(t)$. The induced state equivalence $\simeq^L$ is also called *trace equivalence*.

Example 7.5 [Trace equivalence versus bisimilarity] Consider the observation structure shown in Figure 7.5. The two states s and t are bisimilar, but not trace equivalent. This is because, trace equivalence, unlike bisimilarity, disregards the branching within an observation structure. ■

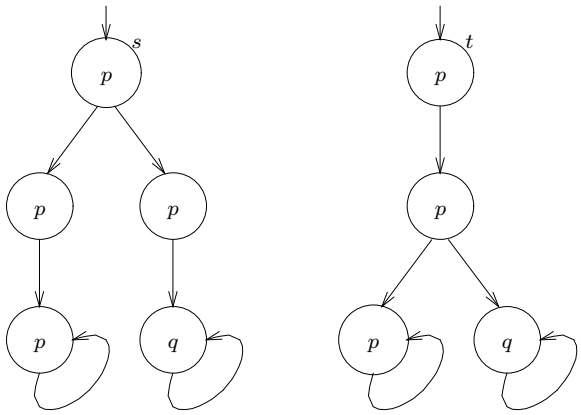


Figure 7.5: Bisimilarity vs. trace equivalence

The next proposition establishes that trace equivalence is less distinguishing than bisimilarity. This means that two bisimilar states are guaranteed to be trace equivalent, but not vice versa. Since bisimilarity is sensitive to the branching nature of the structure, while trace equivalence depends only on the set of trace generated, bisimilarity is called a *branching-time* equivalence, and trace equivalence is called a *linear-time* equivalence.

Proposition 7.2 [Distinguishing power of trace equivalence] *Trace equivalence is less distinguishing than bisimilarity.*

Proof. Let K be an observation structure, and let s and t be two states of K. If s and t are bisimilar, then for every source-s trajectory $\overline{s}_{0\ldots m}$, there exists a source-t trajectory $\overline{t}_{0\ldots m}$ such that $s_i \simeq^B t_i$ for all $0 \leq i \leq m$. Since bisimilar states have identical observations, it follows that every source-s trace is also a source-t trace. Hence, $s \simeq^L t$. This establishes that bisimilarity is as distinguishing as trace equivalence. Example 7.5 establishes that trace equivalence is not as distinguishing as bisimilarity. ■

Like STL, for model checking of SAL formulas it is fine to consider a quotient structure obtained by collapsing states that satisfy the same set of SAL formulas.

Proposition 7.3 [SAL abstraction] SAL *admits abstraction.*

Recall that bisimilarity is a fully abstract semantics for STL: two bisimilar states satisfy the same set of STL formulas, and two non-bisimilar states disagree on the satisfaction of some STL formula. The fully abstract semantics for SAL is trace equivalence: no SAL formula can distinguish between two states that are trace equivalent, and for every two states that are not trace equivalent, there exists a SAL formula that is satisfied by only one of the two states.

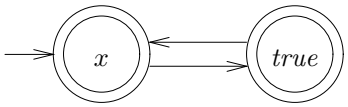


Figure 7.6: The automaton M_{even}

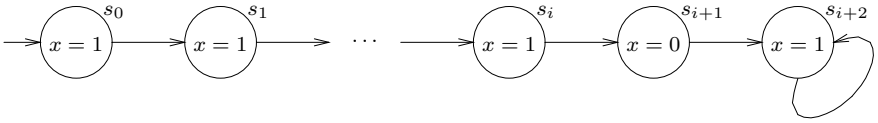


Figure 7.7: STL cannot express $\forall M_{even}$

Theorem 7.1 [Distinguishing power of SAL] *Trace equivalence is a fully abstract semantics for* SAL.

It follows that SAL is less distinguishing than STL.

Corollary 7.1 [Distinguishing power of SAL vs. STL] SAL *is less distinguishing than* STL.

It follows that to solve an instance (K, ϕ) of the model checking problem for SAL, we can construct the minimal quotient $K/_{\simeq^B}$ using one of the algorithms of Chapter 4, and solve $(K/_{\simeq^B}, \phi)$.

7.2.3 The Expressive Power of SAL

If a state logic Φ is more distinguishing than a state logic Ψ, then the logic Ψ cannot be as expressive as Φ.

Example 7.6 [SAL is not as expressive as STL] Consider Figure 7.5. The states s and t are trace equivalent, and hence, satisfy the same set of SAL formulas. On the other hand, for the STL formula $\phi = \forall\bigcirc\exists\bigcirc p$, $s \not\models \phi$ and $t \models \phi$. It follows that no formula of SAL is equivalent to the STL formula ϕ. ■

Even though STL is more distinguishing than SAL, there are requirements that are expressible in SAL, but not in STL.

Theorem 7.2 [Expressive power of SAL vs. STL] *The expressive powers of* SAL *and* STL *are incomparable.*

Proof. We need to establish that STL is not as expressive as SAL. Consider the SAL formula $\phi_{even} = \forall M_{even}$ shown in Figure 7.6, where x is a boolean variable. The SAL formula ϕ_{even} asserts that, given a state s, along every

source-s trajectory x is satisfied in every other state. That is, a state s satisfies ϕ_{even} if for every source-s trajectory $\overline{s}_{0\ldots m}$, for all $0 \leq i \leq m$, if the index i is even, $x[s_i] = 1$.

We wish to establish that no STL formula is equivalent to ϕ_{even}. Assume, to the contrary, that there exists an STL formula ψ such that for every observation structure K, $[\![\psi]\!]_K = [\![\phi_{even}]\!]_K$. Suppose ψ has i subformulas. Consider the observation structure of Figure 7.7. Observe that only one of the two states s_0 and s_1 satisfies ϕ_{even}. We will establish that either both of them satisfy ψ or none of them satisfies ψ.

We prove that for every subformula χ of ψ, if χ has n subformulas, then the states $s_0, \ldots s_{i-n+1}$ agree on the truth of χ. The proof is by induction on n.

Base case $n = 1$: χ is an atomic formula. The states $s_0, \ldots s_i$ have identical observations, and hence, agree on the truth of χ.

Inductive case $n > 1$: We consider the case $\chi = \chi_1 \exists\mathcal{U} \chi_2$, and leave the simpler cases for the reader to verify. Consider two states s_j and s_k with $0 \leq j, k \leq (i-n+1)$. It suffices to prove that if $s_j \models \chi$ then $s_k \models \chi$. Assume that $s_j \models \chi$. We consider the case when $j > k$, and the case $j < k$ is left to the reader.

Suppose $s_j \models \chi_2$. Then, by induction hypothesis, $s_k \models \chi_2$, and hence, $s_k \models \chi$.

Suppose $s_j \not\models \chi_2$. Then there exists $j' > j$ such that $s_{j'} \models \chi_2$ and $s_{j''} \models \chi_1$ for all $j \leq j'' < j'$. Since $s_j \models \chi_1$, by induction hypothesis, $s_{j''} \models \chi_1$ for all $k \leq j'' \leq j$. Hence, $s_k \models \chi$. ■

Exercise 7.5 {T2} [Attempting to specify even requirement in STL] Why is the STL formula

$$x \wedge \forall\Box(x \rightarrow \forall\bigcirc \neg x) \wedge \forall\Box(\neg x \rightarrow \forall\bigcirc x)$$

not equivalent to $\forall M_{even}$? ■

Exercise 7.6 {T3} [Expressive power of SAL vs. STL$^{\mathcal{U}}$] Prove that the expressive powers of SAL and STL$^{\mathcal{U}}$ are incomparable. ■

Exercise 7.7 {T4} [Stutter sensitivity] (1) Show that SAL is stutter-sensitive. (2) Define a stutter-insensitive version of SAL. What is the equivalence induced by stutter-insensitive SAL? How do the distinguishing power and the expressive power of stutter-insensitive SAL compare with STL$^{\mathcal{U}}$? ■

7.3 Operations on Automata

The determinization and product construction for automata are building blocks for solving the language-inclusion problem.

7.3.1 Determinization

In deterministic observation structures, there is at most one initial state per observation, and each state has at most one successor per observation.

DETERMINISTIC OBSERVATION STRUCTURE

Let $K = (G, A, \langle\langle\cdot\rangle\rangle)$ be an observation structure. The observation structure K is *deterministic* if (1) [deterministic initialization] for each observation a of K, there is at most one initial state s with $\langle\langle s\rangle\rangle = a$, and (2) [deterministic update] for each state s and each observation a of K, there is at most one successor t of s with $\langle\langle t\rangle\rangle = a$.

An automaton is deterministic if its observation structure is deterministic. The module P is deterministic if the observation structure K_P is deterministic. The module P is deterministic if (1) it is closed, and (2) the initial commands of all its atoms are deterministic, and (3) the update commands of all its atoms are deterministic.

Example 7.7 [Deterministic structures] The observation structure shown in Figure 6.1 is not deterministic. The SAL formulas of Figure 7.1 and Figure 7.2 are deterministic, while the observation structure of Figure 7.4 is not. ■

Remark 7.4 [Traces define trajectories in deterministic structures] If K is a deterministic observation structure, and $\overline{t}$ and $\overline{u}$ are different trajectories of K with the same source, then $\langle\langle\overline{t}\rangle\rangle \neq \langle\langle\overline{u}\rangle\rangle$. It follows that for each state s of a deterministic observation structure K, there is a one-to-one correspondence between the source-s trajectories and the source-s traces. ■

The region σ of the observation structure K is *consistent* if for all states in σ have the same observation: for all s and t in σ, $s \approx_K t$. In other words, a consistent region is a subset of an $\approx_K$-equivalence class of the propositional equivalence. Every observation structure, and every automaton, can be determinized by replacing the states with the consistent regions. The determinization procedure is usually referred to as the *subset construction*. A state of ΔK is a set of states of K. Intuitively, the transition relation of ΔK is defined so that the sink-state of ΔK corresponding to a trace $\overline{a}$ contains all the sink-states of initialized trajectories of K corresponding to the trace $\overline{a}$. Since our observation structures have finite nonobservable nondeterminism, during determinization, it suffices to consider only finite consistent regions.

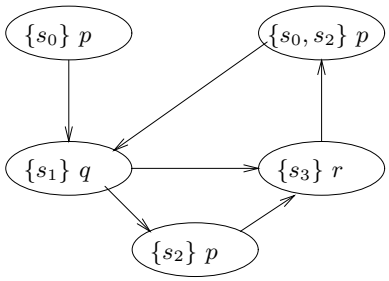


Figure 7.8: Determinized structure

DETERMINIZATION

Let $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\langle\cdot\rangle\rangle)$ be an observation structure. The *determinized structure* ΔK is the observation structure $(\Sigma_\Delta, \sigma^I_\Delta, \rightarrow_\Delta, A, \langle\langle\cdot\rangle\rangle_\Delta)$, where (1) [state space] Σ_Δ is the set of nonempty finite consistent regions of M; (2) [initial region] $\sigma \in \sigma^I_\Delta$ iff there is an observation $a \in A$ such that $\sigma = \{s \in \sigma^I \mid \langle\langle s\rangle\rangle = a\}$; (3) [transition relation] $\sigma \rightarrow_\Delta \tau$ iff there is an observation $a \in A$ such that $\tau = \{s \in post(\sigma) \mid \langle\langle s\rangle\rangle = a\}$; (4) [observation function] $s \in \sigma$ implies $\langle\langle\sigma\rangle\rangle_\Delta = \langle\langle s\rangle\rangle$.

For an automaton $M = (K, \sigma^A)$, the *determinized automaton* ΔM is the automaton $(\Delta K, \sigma^A_\Delta)$, where $\sigma \in \sigma^A_\Delta$ iff $\sigma \cap \sigma^A$ is nonempty.

Example 7.8 [Determinization] Figure 7.8 shows the result of determinizing the observation structure from Figure 6.1. ■

Proposition 7.4 [Determinization] *For every observation structure K, the observation structure ΔK is deterministic and $L_{\Delta K} = L_K$; for every automaton M, the automaton ΔM is deterministic and $L_{\Delta M} = L_M$.*

Exercise 7.8 {T2} [Properties of determinization] Which of the following properties of an observation structure K are inherited by the determinized structure ΔK: serial; finite; finitely-branching; stutter-closed? ■

Observe that the determinized observation structure has exponentially many more states than the original structure, and thus, determinization is a computationally expensive procedure.

Exercise 7.9 {T2} [Cost of determinization] Let K be a finite observation structure with n states and m transitions. Give a tight bound on the number of states and transitions of the determinized structure ΔK. ■

Exercise 7.10 {T3} [Nondeterminism and exponential succinctness] Consider the observation alphabet $A = \{0, 1\}$. For every natural number m, define the language

$$L_m = \{\overline{a}_{0\ldots 2m} \mid \ a_i = a_{i+m} \text{ for some } 0 \leq i < m\}$$

(1) Show that for every natural number m, there is a nondeterministic automaton M with $3m$ states such that $L_M = L_m$. (2) Show that if M is a deterministic automaton with $L_M = L_m$ then M has at least $\mathbf{2}^m$ states. ■

7.3.2 Boolean Operations

Disjoint union

Two observation structures are *disjoint* if their state spaces are disjoint.

DISJOINT UNION

Let $K_1 = (\Sigma_1, \sigma_1^I \rightarrow_1, A_1, \langle\!\langle\cdot\rangle\!\rangle_1)$ and $K_2 = (\Sigma_2, \sigma_2^I, \rightarrow_2, A_2, \langle\!\langle\cdot\rangle\!\rangle_2)$ be two disjoint observation structures. The *union* $K_1 + K_2$ is the observation structure $(\Sigma_1 \cup \Sigma_2, \sigma_1^I \cup \sigma_2^I, \rightarrow_1 \cup \rightarrow_2, A_1 \cup A_2, \langle\!\langle\cdot\rangle\!\rangle)$, where $\langle\!\langle s\rangle\!\rangle = \langle\!\langle s\rangle\!\rangle_1$ if $s \in \Sigma_1$, and otherwise $\langle\!\langle s\rangle\!\rangle = \langle\!\langle s\rangle\!\rangle_2$.

Let $M_1 = (K_1, \sigma_1^A)$ and $M_2 = (K_2, \sigma_2^A)$ be two disjoint automata. The *union* $M_1 + M_2$ is the automaton $(K_1 + K_2, \sigma_1^A \cup \sigma_2^A)$.

Proposition 7.5 [Disjoint union] *If K_1 and K_2 be two disjoint observation structures then $L_{K_1+K_2} = L_{K_1} \cup L_{K_2}$; if M_1 and M_2 are two disjoint automata, then $L_{M_1+M_2} = L_{M_1} \cup L_{M_2}$.*

Remark 7.5 [Cost of union] Let K_1 be a finite structure with n_1 states and m_1 transitions. Let K_2 be a finite structure with n_2 states and m_2 transitions. Assume that K_1 and K_2 are disjoint. Then, the union $M_1 + M_2$ has $n_1 + n_2$ states and $m_1 + m_2$ transitions. Thus, the cost of union is only additive. ■

Product

To obtain intersection of the languages of two observation structures or two automata, we take the cartesian product of the underlying state-spaces.

PRODUCT

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. The *product* $K_1 \times K_2$ is the observation structure $(\Sigma, \sigma^I, \rightarrow, A, \langle\langle\cdot\rangle\rangle)$:

- $\Sigma = \{(s_1, s_2) \mid s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$ and $\langle\langle s_1\rangle\rangle_1 = \langle\langle s_2\rangle\rangle_2\}$;
- $(s_1, s_2) \in \sigma^I$ iff $s_1 \in \sigma_1^I$ and $s_2 \in \sigma_2^I$;
- $(s_1, s_2) \rightarrow (t_1, t_2)$ iff $s_1 \rightarrow_1 t_1$ and $s_2 \rightarrow_2 t_2$;
- $\langle\langle(s_1, s_2)\rangle\rangle = \langle\langle s_1\rangle\rangle_1 = \langle\langle s_2\rangle\rangle_2$.

Let $M_1 = (K_1, \sigma_1^A)$ and $M_2 = (K_2, \sigma_2^A)$ be two automata. The *product* $M_1 \times M_2$ is the automaton $(K_1 \times K_2, \sigma^A)$, where $(s_1, s_2) \in \sigma^A$ iff $s_1 \in \sigma_1^A$ and $s_2 \in \sigma_2^A$.

Proposition 7.6 [Product] *If K_1 and K_2 are two observation structures, then $L_{K_1 \times K_2} = L_{K_1} \cap L_{K_2}$. If M_1 and M_2 are two automata, then $L_{M_1 \times M_2} = L_{M_1} \cap L_{M_2}$.*

Remark 7.6 [Cost of product] Let K_1 be a finite structure with n_1 states and m_1 transitions. Let K_2 be a finite structure with n_2 states and m_2 transitions. Then, the product $K_1 \times K_2$ has at most $n_1 \cdot n_2$ states and at most $m_1 \cdot m_2$ transitions. Thus, the cost of product is multiplicative. ■

Exercise 7.1 shows that the class of languages of observation structures is precisely the class of prefix-closed languages. Since the languages of observation structures are closed under union and intersection, the union and the intersection of two prefix-closed languages are prefix-closed.

Exercise 7.11 {T2} [Operations on automata] Which of the following properties of two observation structures K_1 and K_2 are inherited by the disjoint union $K_1 + K_2$ and the product $K_1 \times K_2$: reflexive; finite; finitely-branching; stutter-closed; deterministic? Which laws govern the interplay of the operations stutter closure $\cdot^S$, determinization Δ, union $+$, and product $\times$ on observation structures? ■

Symbolic representation

Let P be a module. Then, the corresponding observation structure K_P is symbolically represeted by (1) the set X_P of variables, (2) the set $\mathsf{obs}X_P$ of observable variables, (3) the initial predicate q_P^I, and (4) the transition predicate q_P^T. If P and Q are two modules, then the union $K_P + K_Q$ has the initial predicate $q_P^I \vee q_Q^I$ and the transition predicate $q_P^T \vee q_Q^T$, and the product $K_P \times K_Q$ has the

initial predicate $q_P^I \wedge q_Q^I$ and the transition predicate $q_P^T \wedge q_Q^T$. Thus, the union and product operations are easily implemented on the symbolic representations.

Exercise 7.12 {P2} [Symbolic determinization] Write an algorithm that constructs, given the symbolic representation of an observation structure K, the symbolic representation of the determinized structure ΔK. ■

7.3.3 Complementation

Let M_1 and M_2 be two automata with the same set A of observations. The automaton M_2 is a *complement* of M_1 if $L_{M_2} = A^+ \backslash L_{M_1}$. Complementing a deterministic automaton involves two steps: completion and inversion.

In a deterministic automaton, there is at most one initial state per observation, and each state has at most one successor per observation. In a complete automaton, on the other hand, there is at least one initial state per observation, and each state has at least one successor per observation.

COMPLETE AUTOMATON

Let $M = (\Sigma, \sigma^I, \rightarrow, A, \langle\langle\cdot\rangle\rangle, \sigma^A)$ be an automaton. The automaton M is *complete* if (1) $\langle\langle\sigma^I\rangle\rangle = A$ and (2) for each state s, $\langle\langle post_M(s)\rangle\rangle = A$.

An incomplete automaton can be completed, without changing its language, by adding a dummy nonaccepting state for each observation.

COMPLETION

Let $M = (\Sigma, \sigma^I, \rightarrow, A, \langle\langle\cdot\rangle\rangle, \sigma^A)$ be an automaton. The *completion* ΓM is the automaton over the alphabet A with the following components

- every state of M is a state of ΓM, and in addition, for every observation $a \in A$, ΓM has the state s_a;
- every initial state of M is an initial state of ΓM, and in addition, for every observation $a \in A$ such that $a \notin \langle\langle\sigma^I\rangle\rangle$, the state s_a is an initial state of ΓM.
- every transition of M is a transition of ΓM, for all observations $a, b \in A$, ΓM has a transition from the state s_a to the state s_b, and for every state s of M and an observation $a \in A$ such that $a \notin \langle\langle post_M(s)\rangle\rangle$, ΓM has a transition from the state s to the state s_a;
- for $s \in \Sigma$, the observation of s in ΓM is $\langle\langle s\rangle\rangle$, and for an observation $a \in A$, the observation of s_a in ΓM is a.
- the accepting region of ΓM equals σ^A.

Proposition 7.7 [Automaton completion] *For every automaton M, the completion ΓM is complete and $L_{\Gamma M} = L_M$.*

If an automaton M is deterministic, then so is the automaton ΓM. The second step of complementation corresponds to inverting the accepting region.

> INVERSION
> Let $M = (K, \sigma^A)$ be an automaton with the state space Σ. The *inversion* $-M$ is the automaton $(K, \Sigma \backslash \sigma^A)$.

Inversion corresponds to complementation for complete deterministic automata.

Proposition 7.8 [Automaton inversion] *Let M be a deterministic and complete automaton. Then the inversion $-M$ is a complement of M.*

Consequently, determinization, followed by completion, and then by inversion results in complementation.

Corollary 7.2 [Automaton complementation] *For an automaton M, the automaton $-\Gamma\Delta M$ is a complement of M.*

Exercise 7.13 {P1} [Automaton complementation] Apply the completion and inversion operations to the determinized structure of Figure 7.8, and verify that the language of the resulting automaton is the complement of the language of the observation structure of Figure 6.1. ■

Exercise 7.14 {T2} [Properties of complementation] Which of the following properties of a deterministic automaton M are inherited by the complement $-\Gamma M$: serial; finite; finitely-branching; stutter-closed? ■

As mentioned earlier, the class of languages defined by observation structures is not closed under complementation, and thus, complementing an observation structure results in an automaton.

7.4 Model Checking

Now we are ready to address the language-inclusion problem for automata, and consequently, the model checking question for SAL. To check whether the language one automaton M_1 is contained in that of another automaton M_2, it suffices to check whether the language of M_1 has an empty intersection with the complement of the language of M_2.

Proposition 7.9 [Automaton language inclusion] *Let M_1 and M_2 be two automata. Then $L_{M_1} \subseteq L_{M_2}$ iff the language $L_{M_1 \times -\Gamma\Delta M_2}$ is empty.*

Since we already know how to construct the automaton $M_1 \times -\Gamma\Delta M_2$ from the automata M_1 and M_2, we have reduced the language-inclusion problem to that of checking language emptiness. Checking emptiness of the language of an automaton is a reachability problem.

Proposition 7.10 [Automaton language emptiness] *Let M be an automaton with the transition graph G and the accepting region σ^A. Then L_M is empty iff the answer to the reachability problem (G, σ^A) is* NO.

This immediately suggests a decision procedure for the language-inclusion problem. Algorithm 7.1 solves the language-inclusion problem using a depth-first-search on the state space of the product $M_1 \times \Delta M_2$. The completion and inversion steps are applied only implicitly. The abstract data type for automata supports, in addition to the operations *InitQueue* and *PostQueue*, the operation *Accept*, that, given an automaton M returns the accepting region of M, and the operation *Obs*, that, given an automaton M and a state s of M returns the observation of s.

Theorem 7.3 [Language inclusion] *Let M_1 be a finite automaton with n_1 reachable states and m_1 reachable transitions, and let M_2 be a finite automaton with n_2 states. Algorithm 7.1 solves the language-inclusion problem (M_1, M_2) in time $O((n_1 + m_1) \cdot \mathbf{2}^{n_2})$.*

Remark 7.7 [Space complexity of language inclusion] Solving the language-inclusion problem (M_1, M_2) requires searching the state-space of $M_1 \times -\Gamma\Delta M_2$, which can be performed in space logarithmic in the number of states of $M_1 \times -\Gamma\Delta M_2$. Consequently, the language-inclusion problem (M_1, M_2) can be solved in space $O(n_2 \cdot \log\ n_1)$. Thus the space complexity of the language-inclusion problem is PSPACE. The problem is PSPACE-hard in the number of states of K_2. The language-inclusion problem (K_1, K_2) for observation structures is PSPACE-complete in the number of states of K_2. ■

Exercise 7.15 {P2} [Witness reporting in language inclusion] If the answer to the language inclusion problem (M_1, M_2) is NO, then a witness trace is a trace in $L_{M_1} \backslash L_{M_2}$. Modify Algorithm 7.1 so that when it answers NO, it also returns a witness trace. ■

SAL model checking

Consider the model checking problem $(K, \forall M)$, where M is a specification automaton. As discussed earlier, the model checking problem $(K, \forall M)$ can be solved by solving the language-inclusion problem (K, EM). Suppose M has n states and the observations of M are boolean expressions over a set X of propositional variables. According to the definition of EM, a state of EM consists of

Algorithm 7.1 [Language inclusion]

Input: two automata M_1 and M_2.
Output: the answer to the language-inclusion problem (M_1, M_2).

s, t: **state**; σ, τ: **region**;
σ^R: **set of state** $\times$ **region**;
F: **stack of state** $\times$ **region**;

$F := \mathit{EmptyStack}$;
$\sigma^R := \mathit{EmptySet}$;
foreach s **in** $\mathit{InitQueue}(M_1)$ **do**
  $\sigma := \mathit{EmptySet}$;
  foreach t **in** $\mathit{InitQueue}(M_2)$ **do**
    if $\mathit{Obs}(s, M_1) = \mathit{Obs}(t, M_2)$ **then** $\sigma := \mathit{Insert}(t, \sigma)$ **fi**
    od;
  if $\mathit{IsMember}(s, \mathit{Accept}(M_1))$ **and** $\mathit{IsEmpty}(\sigma \cap \mathit{Accept}(M_2))$
    then return NO **fi**;
  if not $\mathit{IsMember}((s, \sigma), \sigma^R)$ **then**
    $F := \mathit{Push}((s, \sigma), F)$;
    $\sigma^R := \mathit{Insert}((s, \sigma), \sigma^R)$
    fi
  od;
while not $\mathit{IsEmpty}(F)$ **do**
  $(s, \sigma) := \mathit{Top}(F)$;
  $F := \mathit{Pop}(F)$;
  foreach t **in** $\mathit{PostQueue}(s, M_1)$ **do**
    $\tau := \mathit{EmptySet}$;
    foreach u **in** $\mathit{PostQueue}(\sigma, M_2)$ **do**
      if $\mathit{Obs}(t, M_1) = \mathit{Obs}(u, M_2)$ **then** $\tau := \mathit{Insert}(t, \tau)$ **fi**
      od;
    if $\mathit{IsMember}(t, \mathit{Accept}(M_1))$ **and** $\mathit{IsEmpty}(\tau \cap \mathit{Accept}(M_2))$
      then return NO **fi**;
    if not $\mathit{IsMember}((t, \tau), \sigma^R)$ **then**
      $\sigma^R := \mathit{Insert}((t, \tau), \sigma^R)$;
      $F := \mathit{Push}((t, \tau), F)$
      fi
    od
  od;
return YES.

Figure 7.9: Checking language inclusion

a state of M and a valuation for the variables in X. Thus, the automaton EM has $n \cdot \mathbf{2}^k$ states if X has k propositional variables. However, there is no need to construct EM explicitly. As usual, we will use on-the-fly representation, that is, implement the functions *InitQueue* and *PostQueue*. Verify that during the execution of Algorithm 7.1, a state (s, t) of EM will be visited only if t is an observation of some reachable state of K. Secondly, during determinization of the automaton EM, a consistent region of EM contains at most n states, and the automaton ΔEM has at most $\mathbf{2}^n \cdot \mathbf{2}^k$ states. Thus, the number of consistent subsets is exponential in the size of the specification automaton M, rather than exponential in the size of the expansion EM. These two observations lead to the following bound on the SAL model checking.

Theorem 7.4 [SAL model checking] *Let K be a finite observation structure with n reachable states and m reachable transitions, and let M be a finite* SAL *specification automaton with k states. Algorithm 7.1 solves the model-checking problem $(K, \forall M)$ in time $O((n + m) \cdot \mathbf{2}^k)$.*

Remark 7.8 [Space complexity of SAL model checking] The SAL model checking problem (K, ϕ) is PSPACE-complete in the size of ϕ. ■

While the cost of SAL model checking is high compared to STL model checking, this is so only in terms of the size of the specification. Since, in practice, the size of the observation structure is a computational bottleneck, while specifications are small, it is fruitful to note the *model complexity* of the model checking problem, namely, the parametric complexity in terms of the size of the structure assuming constant-size specifications.

Remark 7.9 [Model complexity of SAL model checking] Algorithm 7.1 yields a solution to the SAL model checking problem (K, ϕ) with linear-time model complexity. This coincides with the model complexity of the enumerative STL model checking algorithms of Chapter 5. The model complexity of SAL model checking, as well as STL model checking, is NLOGSPACE. ■

SAL verification

The SAL verification problem $(P, \forall M)$ is propositional if P is a propositional module, M is a finite automaton, and the observations of M are propositional formulas. Algorithm 7.1 can be used to solve the SAL verification problem $(P, \forall M)$ for an automaton specification M whose observations are boolean expressions over $\mathsf{obs}X_P$. First, we construct on-the-fly representations of the observation structures K_P and EM, and then, use Algorithm 7.1 to test language inclusion. The complexity is exponential in both the number of variables of P and the number of states of M. As usual, the verification problem is exponentially harder than the corresponding model checking problem.

Remark 7.10 [Space complexity of SAL verification] The SAL verification problem $(P, \forall\phi)$ is PSPACE-complete both in the size of the description of P and in the number of states of ϕ. ■

Deterministic specifications

The exponential complexity of the test for language-inclusion (M_1, M_2) disappears if M_2 is deterministic. When M_2 is deterministic, then Algorithm 7.1 solves the language-inclusion problem (M_1, M_2) in time $O((n_1 + m_1) \cdot n_2)$. Consequently, the SAL model checking problem $(K, \forall M)$ is computationally easier when the expansion EM is deterministic. The expansion EM is guaranteed to be deterministic if

1. the observations of every pair of initial states of M are mutually exclusive (i.e. if s and t are initial states, the observation of s is the expression p, and the observation of t is the expression q, then the conjunction $p \wedge q$ is an unsatisfiable formula), and
2. for every state s of M, the observations of every pair of successor states of s are mutually exclusive.

In such a case, the complexity of SAL model checking $(K, \forall M)$ is $O(|K| \cdot |M|)$.

There is another advantage of restricting to deterministic specification automata: the language-inclusion problem (M_1, M_2) can be solved symbolically if the automaton M_2 is deterministic.

Exercise 7.16 {P3} [Symbolic language inclusion] Consider the language-inclusion problem (K_1, K_2) when the observation structure K_1 is represented symbolically, and the observation structure K_2 is represented enumeratively. Write an algorithm for solving the language-inclusion problem that combines symbolic reachability analysis of K_1 with determinization of K_2. ■

Complemented specifications

Example 7.3 shows two ways of writing a requirement in SAL. A mutual-exclusion protocol satisfies the exclusion requirement if it satisfies the SAL formula $\forall M_{mutex}$, or equivalently, the formula $\neg\exists M_{\neg mutex}$. Checking whether an observation structure satisfies $\forall M_{mutex}$ requires complementing M_{mutex}, and hence, determinizing it. On the other hand, an observation structure satisfies $\neg\exists M_{\neg mutex}$ if the product structure $K \times EM_{\neg mutex}$ has an empty language, and thus, can be checked without determinization. This holds whenever the SAL specification is given as a negation of an automaton that accepts the undesirable behaviors.

Proposition 7.11 [Checking negated automata] *The answer to the* SAL *model checking problem* $(K, \neg\exists M)$ *is* YES *iff the language* $L_{K \times EM}$ *is nonempty.*

Consequently, the SAL model checking problem $(K, \neg\exists M)$ can be solved in time $O(|K| \cdot |M|)$ by exploring the product of K and EM on-the-fly. Thus, requiring the user to specify the automaton accepting the undesirable behaviors, rather than the automaton accepting desirable behaviors, simplifies the model checking task.