

Chapter 4

Systems

4.1 Introduction

The previous examples gave an informal introduction to SESF. We now present the theory underlying SESF, starting with systems in this chapter. Later chapters cover assertions, proof rules, services, service satisfaction, and compositionality. Below, we cover system programs, systems consisting of one process, systems consisting of multiple processes, and finally, the interleaving semantics of systems.

4.2 System programs

A system program has the structure:

```
system-program <name>( <parameters> ) { // header
  <functions> // includes initialization function
  <named lc events> //
  <externally-controlled events> // xc events
  <fairness assumptions>
}
```

The header indicates the system program's name and any parameters and their types. Functions are as in any procedural language; they can define and update variables, call functions, create processes, block on synchronization constructs, and so on. There is usually an `init()` function for initializing variables and start processes. Named `lc` events are a notational convenience explained later. Externally-controlled events are the functions that can be called by the environment. The fairness assumptions define the fairness expected of the underlying platform in scheduling processes of the system. We use the abbreviations `xc` for externally-controlled and `lc` for locally-controlled.

Functions

Functions have the form

```
function <return type> <name >( <input parameters> ) {
  <body>
}
```

The function header indicates the **function's signature**, consisting, in general, of the return type, the function name, and the function parameters and their types. The return type is absent if the function does not return a value. The parameters are read-only and they can be absent.

We use the construct `StartSystem(M)` to create a process and have it start executing a system program `M` (with any parameters instantiated). We use the construct `StartThread(f)` to create a thread and have it

start executing a function f of this system program (with any parameters instantiated). A thread shares the address space of the process that executed it, whereas a system has its own address space.

The function body can call xc events of *other* systems. An **event call** has the syntax $P.e(x)$, where P is the name of the system whose xc event is being called, e is the event name, and x are the instantiated parameters if any. Note that an event call is very different from a function call. A function call involves the traditional sequential transfer of control within the same system, whereas an event call involves the (possibly concurrent) execution of code in another system (accessed perhaps by an interprocess interaction such as TCP/IP, remote procedure invocation, or http).

Atomicity in the function body is indicated by enclosing atomically-executed code in angled brackets, as in $\langle S \rangle$, or by an explicit statement such as “every memory read and memory write is atomic”. An atomically-executed code fragment corresponds to a locally-controlled event, or **lc event** for short. It can make at most one event call in any execution. A lc event is said to be **enabled** if a process is at the event and the event, if it has a blocking condition, is not blocked. For example, a semaphore wait statement is enabled if a process is at the statement and the semaphore has a nonzero value.

Nondeterministic code is allowed, and is sometimes essential. In particular, we denote nondeterministic selection by \square . When the construct $S \square T$ is executed, either S or T is selected arbitrarily and executed. The \square operator can also be a quantifier; for example, $\{ \square Y x :: S(x) \}$ executes $S(x)$ with parameter x set to an arbitrary value from domain Y .

Externally-controlled events

Externally-controlled events, or **xc events** for short, are like functions with special features. They are the only parts of the system program that are externally visible. They can be called only from the environment. They are executed atomically and without blocking provided the call is “safe” (explained below). They can do a very restricted form of event call, as discussed next.

Allowing xc events to do event calls opens up the possibility of event call chains and cycles, for example, $P.e$ calls $Q.f$ which calls $R.g$ which calls $P.h$. This opens up a can of worms, requiring us to define what it means for an event’s atomic execution to be interrupted by another event’s execution, as in $P.h$ executing within $P.e$ ’s execution. On the other hand, disallowing event calls in xc events is too restrictive. It eliminates “read-modify-write” atomic interactions between systems, which includes a very important class of synchronization primitives.

Our approach is to allow xc events to do a very restricted form of event call, specifically, return a value. An xc event returning a value is effectively doing an event call to the system that called the xc event, but this is a special case that does not give rise to a call cycle because the return value is absorbed by the lc event that called the xc event in the first place.

An xc event has the form

```
xc-event <return type> <event name>( <event input parameters> ) {
  ec <enabling condition predicate> // not checked by system, no side effects
  ac <action> // no event calls, no blocking
}
```

The header indicates the **event’s signature**, similar to a function’s signature, consisting of return type (absent if the event does not return a value), event name, and event input parameters (if any) and their types. The **enabling condition** is a predicate in the program variables and parameters. We say an event call is **enabled** if the event’s enabling condition holds for the parameters (if any) of the call. The **action** is the code that is executed when the event is called. It has no event call. It returns a value iff the return type is present. We refer to an xc event as **xc-without-return** or **xc-with-return**, depending on whether or not it returns a value.

We next define the notion of safe event calls and safe event returns. An event call $P.e(x)$ is **safe** if (1) it is **signature-consistent**, that is, system P exists, has e as an xc event, and the instantiated parameters x match the event’s signature, and (2) the enabling condition of $e(x)$ holds when the call is made. For a call $e(x)$ of an xc-with-return event, the return is **safe** if the value returned is of the return type.

It is the caller’s responsibility, not the callee’s, to ensure that the call is safe. The caller must determine

this based solely on the event's signature and past interaction with the callee, since nothing else of the callee system is visible. *For a safe event call, the callee's responsibility is to execute the action atomically without blocking and, for an xc-with-return event, to do a safe return.* There is no obligation on the callee if the call is not safe. The callee is not obliged to check that the call is safe, but it can choose to do so in the action (presumably, signalling an error if the call was not safe).

We have imposed the above requirement that a safe event call be nonblocking because it simplifies the theory without any loss of generality. A blockable input operation, e.g., a semaphore wait operation, would be modeled in our formalism by two events, an xc event corresponding to initiating the operation, and a lc event corresponding to the return of the operation. This does not introduce more complexity; it merely makes explicit the inherent complexity of blockable input operations.

Fairness assumptions

The fairness assumptions define the fairness expected of the underlying platform in scheduling the processes of the system program. It has the structure

```
fairness-assumptions { <fairness assertions> }
```

We use the following notation for fairness assertions, where p is a process or thread of the system and x is an lc event of the system (note that a process is always at some lc event). $Wfair(p, x)$ denotes weak fairness for p at x , i.e., if p is at x and any blocking condition of x is continuously enabled, then p eventually executes x . $Sfair(p, x)$ denotes strong fairness for p at x , i.e., if p is at x and any blocking condition of x is continuously or intermittently enabled, then p eventually executes x . If a parameter is omitted in a fairness assertion, then it means that the fairness assertion holds for all values of the parameter. So $Wfair(x)$ means $Wfair(p, x)$ for every p , $Wfair(p)$ means $Wfair(p, x)$ for every x , and $Wfair()$ means $Wfair(p, x)$ for every p and x . $Sfair(x)$, $Sfair(p)$, and $Sfair(*)$ are similarly defined. Usually one expects an execution platform to provide $Wfair()$ and $Sfair(x)$ for particular lc events x , for example, semaphore wait operations.

Named locally-controlled events

A named locally-controlled event, or **named lc event**, has the structure:

```
lc-event <event name>(<event parameters>)
  ec  <enabling condition>    // checked by system, no side effects
  ac  <action>                 // can have event calls
```

The header indicates the event name and any parameters and their types. There is no return type. The enabling condition, as in xc events, is a predicate in program variables and parameters, except that here it is checked by the system. Whenever the event is enabled, the action can be executed. The action should execute atomically and without blocking; thus the enabling condition is the only place to block the event. The action can have event calls; as usual, at most one event can be called in an execution.

The named lc event construct is just a way to specify activity without explicitly creating processes. Although not present in most conventional programming languages (e.g., Java), it is easily implemented, for example, by having a thread repeatedly atomically check the enabling condition and execute the action whenever it holds.

Consider a system program in which locally-controlled activity is defined entirely by named lc events. Its code has no process creation or blocking construct. Atomicity is indicated by the events rather than within function bodies (a function is executed only when called within an event). Progress assumptions are stated for lc events (and not processes). Essentially, *such a program defines what the system does without identifying its processes.* Such programs are ideal for defining services and during system design, whereas processes would introduce needless structure and complications. Processes are really an implementation issue, arising because that is how most programming languages and operating systems support concurrency. The environment of a system does not care which particular process in the system is responsible for doing an output.

4.3 Basic and composite systems

A system comes into existence when one starts executing a system program. One can also treat *any* collection of systems as a system, for example, a collection of systems spread over different sites as a single distributed system. We refer to a system as **composite** if it contains other systems; these other systems are referred to as **component systems** of the composite system. We refer to a system as **basic** if it contains no other system.

The life of a system starts with initialization, proceeds via a succession of lc and xc event executions, and ends if and when the system has no more code to execute or experiences a fault. An event execution can be an interaction with the environment or an interaction within the system. The latter, referred to as an **internal interaction**, happens when the system executes an lc event that either does not call an event or calls an event within the system (i.e., the system is composite, and the calling and called events belong to different components). Figure 4.1 illustrates an execution of a basic system with one process. Figure 4.2 illustrates an execution of a composite system consisting of three component systems, each with a single process.

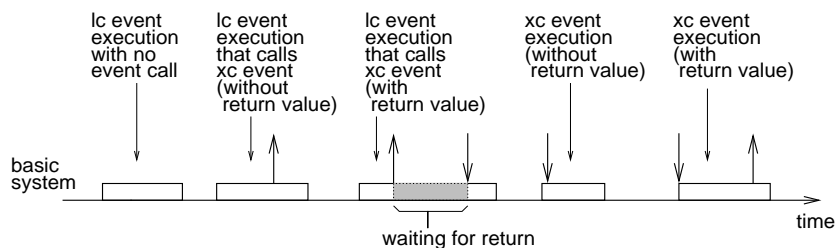


Figure 4.1: An evolution of a basic single-process system.

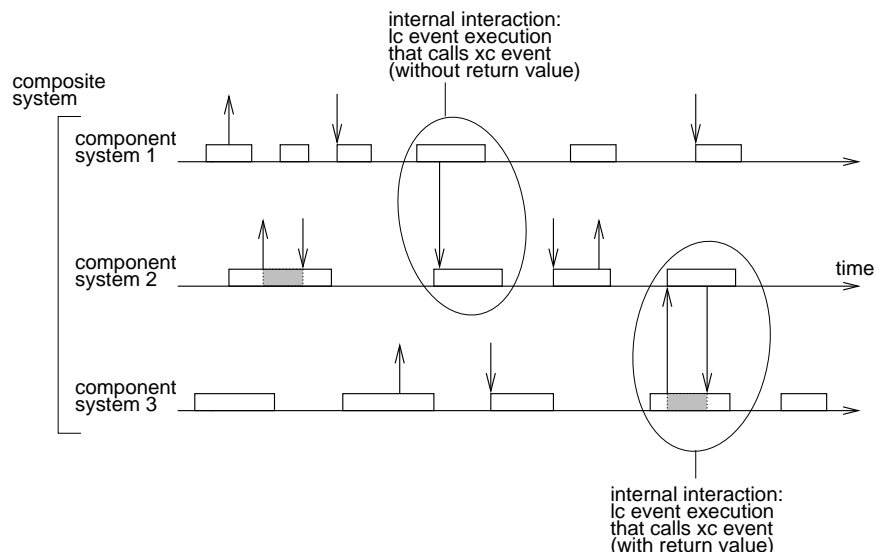


Figure 4.2: An evolution of a composite system of three single-process systems.

Faults are classified into locally caused and externally caused. A **locally-caused fault** happens if the system executes an undefined or nonterminating operation, for example, division by zero, dereferencing a null pointer, signature-invalid call or return, infinite loop, etc. An **externally-caused fault** happens if the environment makes an unsafe call of an event of the system or an unsafe return in an xc-with-return event call made by the system. An internal interaction of a composite system in which the called xc event is not enabled is classified as a locally-caused fault of the system.

A faulty system can behave *arbitrarily*, including crashing, blocking, or behaving abnormally later on. *We do not care what happens after a system becomes faulty, but we do care who caused the first fault.* We expect a system to be free of locally-caused faults, but of course, this is just an expectation.

The **IO signature** of a system refers to the signatures of (1) the xc events of the system that are visible to the environment, and (2) the xc events in the environment that the system can call. We denote the first set of xc events as **inputs** of the system, and the second set of xc events as **outputs** of the system. By default, all xc events of the system are visible, but this can be overridden to hide some xc events from the environment. An empty IO signature means that the system is **closed**: it does not interact with its environment. The default for a composite system is that the inputs consist of the inputs of the components, and the outputs consist of the outputs of the components that are not inputs of the components.

Recall that an execution of a composite system is made up of executions of its component systems stitched together at matching interactions. The converse also holds: any collection of component system executions that make “compatible” sequence of calls to each other can be stitched together to form an execution of the composite system. More precisely, let P be a composite system with component systems P_1, \dots, P_N . Let x_1, \dots, x_N be executions of P_1, \dots, P_N . Let $x_i[P]$ be the sequence of calls and returns of xc events of P in x_i . We say the executions $\{x_1, \dots, x_N\}$ are **signature-compatible** if the sequences $\{x_1[P], \dots, x_N[P]\}$ can be merged such that for every $i, j, i \neq j$, (1) every call of an xc event of P_j from $x_i[P]$ is followed immediately by an identical call from $x_j[P]$, and (2) every return of an xc event of P_j from $x_j[P]$ is followed immediately by an identical return from $x_i[P]$. We show below, in Theorems 4.2 and 4.1, that the executions $\{x_1, \dots, x_N\}$ can be stitched together to form an execution z of P if and only if $\{x_1, \dots, x_N\}$ are signature-compatible.

4.4 Interleaving semantics of basic systems

To reason about a system, we need a mathematical model of its executions. Ours is based on nondeterministic interleaving, where an execution of a system is represented by a succession of atomic event executions, with at most one event undergoing execution at any time. This section gives the model for a basic system in terms of the program it executes.

Every possible value assignment to the variables and parameters of the system program defines a **state** of the system. If the program has processes, then their control pointers are also included in the variables. The set of variables and parameters is, in general, dynamic. States where an event is undergoing execution, i.e., a process control is inside the event, are referred to as **intra-event states**. States where no event is undergoing execution are referred to as **inter-event states**. The **initial state** of the program refers to the inter-event state resulting after the execution of the program’s initialization code. We treat a system that has experienced a fault as having entered a “faulty” state, denoted by the symbol **fault**.

There is a key difference between intra-event states and inter-event states, one that goes to the heart of observability in concurrent systems. *An intra-event state of an event execution is observable only to the process executing the event and to the (human or mechanical) analyst.* The intra-event state is invisible to every other process in the world. Consequently, the behavior of the system and its interaction with the environment is fixed by the sequence of inter-event states and events traversed during execution. Intra-event states are only relevant for their part in determining the inter-event states.

Every named event of the system defines a set of **event calls**, one for each possible value of the event’s parameters. The set has a single entry if the event has no parameters. Every xc-with-return event also defines a set of **event returns**, one for each possible value of the return type. Let **input calls** denote the event calls of the inputs of the system. Let **output calls** denote the event calls of the outputs of the system.

Transitions model event executions, specifically, the executions of lc events and safe xc event calls. A transition is simply the sequence of states, event calls, and event returns traversed during an event execution. A transition starts in an inter-event state and ends either in an inter-event state if no fault is encountered or the symbol **fault** if a fault is encountered. Transitions are classified into internal, output, and input as follows.

- **Internal transitions** represent executions of lc events in which no event is called. A fault-free internal transition involving a named lc event has the form $\langle s, e, x_1, x_1, x_2, \dots, x_n, t \rangle$ where e is the lc event call, s is an inter-event state where the lc event is enabled, x_1, x_2, \dots, x_n is the (perhaps empty) sequence

of intra-event states traversed during the execution, and t is the terminating inter-event state. Using \bar{x} to denote the sequence x_1, x_2, \dots, x_n , this transition has the form $\langle s, e, \bar{x}, t \rangle$. For an lc event without a name (e.g., in a process-only system), the lc event call e is absent.

A faulty internal transition (i.e., one where a fault occurs) has the same form as above except that it ends in **fault** at the intra-event state where the fault occurs; thus it has the form $\langle s, e, \bar{x}, \text{fault} \rangle$.

- **Output transitions** represent lc event executions in which an event is called. A fault-free transition involving the execution of a named lc event and an output call of an xc-with-return event has the form $\langle s, e, \bar{x}, f, f', \bar{z}, t \rangle$ where e is the lc event call, s is an inter-event state where the lc event call is enabled, \bar{x} is a intra-event state sequence occurring prior to the output call, f is a call of the xc-with-return event, f' is f 's return and can be any value consistent with f 's signature, \bar{z} is a intra-event state sequence following the return, and t is the terminating inter-event state. If the lc event has no name, then e is absent. If the call f is to an xc-without-return event, then f' is absent (and \bar{z} is the intra-event state sequence after the call).

A faulty output transition has the same form as above except that it ends in **fault** at the point where the fault happens. Note that the fault can be an undefined local operation or an undefined output call (e.g., call of non-existent event, signature-inconsistent call).

- **Input transitions** represent xc event executions: A fault-free transition involving an xc-with-return event has the form $\langle s, e, \bar{x}, e', \bar{y}, t \rangle$ where e is a call of an input (i.e., an xc event visible to the environment), s is an inter-event state where the event is enabled, \bar{x} is a intra-event state sequence occurring prior to the return value, e' is the returned value, \bar{y} is a intra-event state sequence occurring after the return value, and t is the terminating inter-event state. For an xc-without-return event, e' and \bar{y} are absent.

A faulty input transition is as above except that it ends in **fault** at the point where the fault happens. The fault can be caused by the system, as in the case of an undefined local operation or a signature-inconsistent return (if the input is of an xc-with-return event). The fault can also be caused by the environment making an unsafe input call; in this case the transition has the form $\langle s, e, \text{fault} \rangle$ where e is a signature-inconsistent call or s does not satisfy the call's enabling condition.

The transitions above may come from deterministic or nondeterministic events. A nondeterministic event can give rise to several different transitions starting from the same system state s . For example, a nondeterministic lc event call e may have the transitions $\langle s, e, \bar{x}_1, \text{fault} \rangle$, $\langle s, e, \bar{x}_2, t_2 \rangle$, $\langle s, e, \bar{x}_3, f_3, \bar{y}_3, t_3 \rangle$, $\langle s, e, \bar{x}_4, f_4, f'_4, \bar{y}_4, t_4 \rangle$. Similarly, a nondeterministic xc event call e may have the transitions $\langle s, e, \bar{x}_1, \text{fault} \rangle$, $\langle s, e, \bar{x}_3, e'_3, \bar{y}_3, t_3 \rangle$, $\langle s, e, \bar{x}_4, e'_4, \bar{y}_4, t_4 \rangle$.

An **execution** of the system is a sequence of transitions starting from the initial state. Executions are of three kinds. **Fault-free finite executions** are sequences of the form $\langle s_0, \bar{p}_0, s_1, \bar{p}_1, s_2, \dots, s_n, \bar{p}_n, s_{n+1} \rangle$ where s_0 is an initial state, and every $\langle s_i, \bar{p}_i, s_{i+1} \rangle$ is an fault-free transition. **Infinite executions** are fault-free and never end, so they have the form $\langle s_0, \bar{p}_0, s_1, \bar{p}_1, s_2, \bar{p}_2, s_3, \dots \rangle$. **Faulty executions** are finite and end in **fault**. So the execution has the form $\langle s_0, \bar{p}_0, s_1, \bar{p}_1, s_2, \bar{p}_2, s_3, \dots, s_n, \bar{p}_n, \text{fault} \rangle$.

The set of executions of a system represent all the possible evolutions of the system. This set decides the safety properties of the system. We expect a system to have no faulty executions except for those caused by the environment (i.e., unsafe input calls or unsafe event returns of output calls), but this is just an expectation.

Not all the executions satisfy the fairness assumptions of the system. A finite execution satisfies (weak or strong) fairness for an event iff the execution is fault-free and the event is not enabled in the last state of the execution. An infinite execution satisfies weak fairness for an event iff the event occurs infinitely often or is disabled infinitely often (i.e., at an infinite number of inter-event states). An infinite execution satisfies strong fairness for an event iff the event occurs infinitely often if it is enabled infinitely often.

An execution of the system is said to be **complete** iff it satisfies all the fairness assumptions of the system. Intuitively, a complete execution is one where the underlying platform has satisfied the scheduling assumptions of the system. The progress properties of the system are decided by the set of complete executions, and not by the set of all executions.

The set of executions of a system can have faulty executions, whereas the set of its complete executions has only fault-free executions. Also the set of executions of a system is prefix-closed, whereas the set of its complete executions is not prefix-closed (unless the progress assumptions are vacuous).

We refer to the sequence of event calls and event return values in an execution as the **event trace** of the execution. A **complete event trace** is the event trace of a complete execution. As with executions, the event traces of a system are prefix-closed whereas the complete event traces are not prefix-closed in general.

4.5 Semantics of composite systems

We now extend the semantics to composite systems. Consider a composite system C . From the interleaving model, each execution of the composite system is a sequence of atomic event executions. An event execution involves either one component or two components. The former case is when one component does an internal transition or an output or input transition with the composite system's environment; the other components undergo no change. The latter case is when one component executes a lc event which calls an xc event of another component; here, the two components undergo transitions and all other components undergo no change.

The state space of C is the product of the component state spaces. The inter-event state space of C is the product of the component inter-event state spaces. The intra-event state space of C is the state space of C where exactly one event is in the middle of execution. The initial state of C is the product of the component initial states.

We define an **image** mapping from the composite system to a component system as follows:

- For a state s of C and a component system P , the **image of s on P** , denoted by $s.P$, refers to the part of s corresponding to P , i.e., the value assignment to the variables of P .
- For a sequence x of states, event calls, and event returns of C , the **image of x on P** , denoted by $x.P$, refers to the sequence obtained from x by doing the following in order: (1) replace every state s by $s.P$; (2) delete every event call and every event return not in P 's IO signature; (3) replace consecutive occurrences of the same state by a single occurrence.

The input calls of the composite system are the union of the input calls of the component systems. The output calls of the composite system are the union of the output calls of the component systems minus the input calls of other component systems of the composite system.

The transitions of C are derived from the transitions of the component systems. Each transition d of a component system P gives rise to a set of transitions of C . In each transition b of this set, P undergoes the same changes as it does in d while the other components undergo no change, unless d is an output transition that calls an xc event of another component Q , in which case component Q also undergoes changes according to a transition of the called event.

We next restate the above formally, using s, t for C 's inter-event states, x, y, z for C 's intra-event states, $\bar{x}, \bar{y}, \bar{z}$ for C 's intra-event state sequences, e, f for event calls, e', f' for their corresponding event returns if any, and P, Q, R to range over the component systems.

- **Basic internal transitions** of C represent executions of lc events in which no event is called. A (fault-free or faulty) basic internal transition involving a named lc event has the form $\langle s, e, \bar{x}, t \rangle$ such that (1) for some component P , the transition's image on P (which equals $\langle s.P, e, \bar{x}.P, t.P \rangle$) is a (fault-free or faulty) internal transition of P , and (2) for every other component Q , the image of the transition on Q reduces to $\langle s.P \rangle$ where $s.P$ is an inter-event state of Q or fault. For a lc event without a name the lc event call e is absent.
- **Output transitions** of C represent executions of lc events that call events in C 's environment. A (fault-free or faulty) output transition involving a named lc event and an xc-with-return called event has the form $\langle s, e, \bar{x}, f, f', \bar{z}, t \rangle$ such that (1) f is a call of an xc event in the environment of C (i.e., not belonging to a component of C), (2) for some component P , the transition's image on P is a (fault-free or faulty) output transition of P with event call f , and (3) for any other component Q , the transition's image on Q reduces to $\langle s.P \rangle$ where $s.P$ is an inter-event state of Q or fault. If the called event f does not return a value, then f' is absent.

- **Input transitions** of C represent executions of xc events of C 's called by C 's environment. A (fault-free or faulty) input transition involving an xc -with-return event has the form $\langle s, e, \bar{x}, e', \bar{y}, t \rangle$ such that (1) for some component P , the image of the transition is a (fault-free or faulty) input transition of P , and (2) for any other component Q , the image of the transition reduces to $\langle s.P \rangle$ where $s.P$ is an inter-event state of Q or **fault**. For an xc -without-return event, e' and \bar{y} are absent.
- **Composite internal transitions** of C represent executions of lc events in C that call events in other components of C . A fault-free composite internal transition has the form $\langle s, e, \bar{x}, f, \bar{y}, f', \bar{z}, t \rangle$ such that for two components P and Q , (1) the transition's image on P is a fault-free output transition of P in which a call f is made to an xc event of Q , (2) the transition's image on Q is a fault-free input transition of Q , and (3) for any other component R , the transition's image on R reduces to $\langle s.R \rangle$ where $s.R$ is an inter-event state of R or **fault**. If f does not return a value, then f' is not present.

A faulty composite transition of C is as above except that one or both of P and Q enter **fault**, depending on what exactly happens. [For example, Q enters **fault** if f is unsafe (signature-inconsistent or disabled); P enters **fault** if f is xc -with-return and the return is unsafe. Whether one component's fault affects another component depends on the platform's run-time checks.] In any case, the transition's image on P is an output transition of P unless P enters **fault**, the transition's image on Q is an input transition of Q unless Q enters **fault**, and the transition's image on any other component R reduces to $\langle s.R \rangle$ where $s.R$ is an inter-event state of R or **fault**.

This completes the definition of the transitions of the composite system C . The transitions give rise to **executions**, **complete executions**, **event traces**, and **complete event traces** of C , exactly as in the case of a basic program.

Recall that an execution of a composite system is made up of executions of its component systems stitched together at matching interactions. This is formalized in the following theorems.

Theorem 4.1 Let P be a component of a composite system C . Let x be a fault-free execution of C .

- $x.P$ is a fault-free execution of P .
- If x is a complete execution of C then $x.P$ is a complete execution of P .

■

Proof Part (a): The proof is by induction on the number of transitions in x . The theorem holds for x consisting of just an initial state. Suppose it holds for a finite execution x . It suffices to show that if x is extended to z by a transition b of C , then $z.P$ is an execution of P .

Let s be the last state of x . Thus b starts with the inter-event state s . Let b' be b without this leading inter-event state. Then z is $x \circ b'$, where \circ denotes concatenation.

Let b be a simple internal transition of C (e.g., if b is a transition of an lc event named e then b has the form $\langle s, e, \bar{x}, t \rangle$ and b' is $\langle e, \bar{x}, t \rangle$). If transition b arises from an event of P , then $b.P$ is a transition of P and so $z.P$, which equals $x.P \circ b'.P$, is an execution of P (since $x.P$ is an execution of P from the induction hypothesis). If the transition does not arise from an event of P , then $b.P$ reduces to $\langle s.P \rangle$ and $z.P$ equals $x.P$ and is an execution of P .

Let b be an input transition of C . The argument is the same as for internal transition.

Let b be an output transition of C . The argument is the same as for internal transition.

Let b be a composite internal transition of C , where the calling event belongs to component R and the called event belongs to component S . If P is R , then $b.P$ is an output transition of P and so $z.P$, which equals $x.P \circ b'.P$, is an execution of P . If P is S , then $b.P$ is an input transition of P and so $z.P$, which equals $x.P \circ b'.P$, is an execution of P . If P is neither R nor S , then $b.P$ reduces to $\langle s.P \rangle$ and $z.P$ equals $x.P$ and is an execution of P .

Part (b): Let x be a complete execution of C . Thus x is fault-free and $x.C$ is a fault-free execution of P . It suffices to show that, for any lc event e of P , $x.P$ satisfies progress for e iff x satisfies progress for e . That is, we need to show that at every inter-event state s of x , (1) e is enabled at s iff e is enabled at $s.P$; and (2) e 's execution at s is well-formed iff e 's execution at $s.P$ is well-formed. Condition 1 holds because system

composition does not affect the enabling condition of a lc event (in fact, condition 1 holds for any state s of C). Condition 2 holds because $x.P$ is fault-free. ■

Theorem 4.2 Let C be a composite system with components P_1, P_2, \dots, P_N . Let x_i be a fault-free execution of P_i , for $i = 1, \dots, N$.

- (a) There exists a fault-free execution z of C such that $z.P_i = x_i$ for every $i = 1, \dots, N$ if and only if the set of executions $\{x_1, \dots, x_N\}$ is signature-compatible.
- (b) Furthermore, z is a complete execution of C iff every x_i is complete execution of P_i . ■

Proof Part (a)

Case 1: We first consider the “if” direction for $N = 2$, i.e., C consists of P_1 and P_2 . We assume x_1 and x_2 are signature-compatible and show that they can be stitched together to form an execution z of C . Because x_1 and x_2 are compatible, there is a compatible merge of $x_1[C]$ and $x_2[C]$. The proof is by induction on the number of P_1 - P_2 interactions in this merge, where a P_1 - P_2 interaction is a call (and return, if any) of an xc event of one component by the other.

The base case is when there are no P_1 - P_2 interactions in the merge. Then any interleaving of the transitions in x_1 and x_2 , e.g., $x_1 \circ x_2$, yields an execution z of C such that $z.P_1 = x_1$ and $z.P_2 = x_2$.

Now assume that the result holds for upto some number of P_1 - P_2 interactions, and consider executions x_1 and x_2 whose compatible merge has one more P_1 - P_2 interaction. Let x'_1 be the prefix of x_1 ending just before the last P_1 - P_2 interaction. Let x'_2 be the prefix of x_2 ending just before the last P_1 - P_2 interaction. By the induction hypothesis, there is a finite execution z' of C such that $z'.P_1 = x'_1$ and $z'.P_2 = x'_2$. The last P_1 - P_2 interaction is enabled at the end of z' . Executing that interaction (i.e., composite internal transition) takes C to a state from where P_1 can execute the transitions in the suffix of x_1 following x'_1 and P_2 can execute the transitions in the suffix of x_2 following x'_2 . Executing these transitions in any order extends z' to an execution z that satisfies $z.P_1 = x_1$ and $z.P_2 = x_2$.

Case 2: The “only if” direction for $N = 2$ is easy. Given z such that $z.P_1 = x_1$ and $z.P_2 = x_2$, it is obvious that x_1 and x_2 are signature-compatible.

Case 3: The extension to a composite system C with components P_1, P_2, \dots, P_N , where $N \geq 3$, is straightforward and not expanded on here. [One way to do it is apply the result for $N = 2$ to the intermediate composite systems $Q_2 = (P_1, P_2)$, $Q_3 = (Q_2, P_3)$, \dots , $Q_N = (Q_{N-1}, P_N)$.]

Part (b): Similar to part (b) of Theorem 4.1. ■

4.6 Concluding Remarks

Our system model is not restricted to finite state machine models. Most formalisms of such expressive power, including ours, model concurrency by nondeterministic interleaving rather than by partial orders. Because the interleaving approach explicitly supports the notion of a global state of a concurrent system, it is very helpful in reasoning about system behavior.

Our syntax for specifying systems is similar to that of most procedural programming languages, in that the program structure is similar, processes are explicit, and systems interact via function calls. Most other formalisms of similar expressive power use a more abstract syntax for specifying systems, which allows the theory to be notationally simpler [1, 2, 37, 38, 30, 32, 33, 9, 10, 6, 7, 35, 36, 27].

In almost all formalisms, programs have the event-based structure. In some formalisms, systems interact via shared variables rather than function calls (e.g., [1, 2, 30, 32, 33, 10]). In most formalisms where systems interact via function calls, instead of lc events, a system has “internal” events and “output” events. Both types of events have the syntax of an named lc event with no event calls (i.e., enabling condition and action involving only local variables). An output event has the name of an input (or xc) event of another system, denoting that the output event’s execution is coupled with (or calls) the matching input event. Thus output (internal) event names are a subset of (disjoint from) the names of input events in the environment.

Another distinction between the formalisms is in their syntax for specifying event actions. Some formalisms use informal (natural language) syntax (e.g., [35, 36, 37, 38]). Some formalisms use predicates relating the values of “type-free” system variables before and after the event execution [1, 2, 27]. This approach, which is essentially equivalent to using pre- and post-conditions, usually hides the computational cost of the event as well as the presence of nondeterministic selection. Some formalisms, including the one here, use procedural code with assignments (e.g., [10, 6, 7]).

Most approaches, including ours, have the restriction that safe inputs are never blocked. CSP-based formalisms (e.g., [25]) are an important exception to this.

4.7 Exercises

- 4.1 Let X be a system program whose initialization starts a process that executes a non-terminating loop (LockMgr is such a system). Give conditions under which every complete execution of X is an infinite execution.
- 4.2 Let Y be a system program that has no blocking lc-events. Give conditions under which every complete execution of Y is a finite execution.

Chapter 5

Assertions and Proof Rules

5.1 Introduction

Assertions are a convenient formalism for specifying safety and progress properties of system executions. Assertions can be proved by either operational arguments or by proof rule applications. Proof rules are templates that allow one to infer an assertion given the constructs of the system being analyzed and previously established assertions. A proof rule application can be checked without understanding the system, and hence such proofs are usually very trustworthy. The flip side is that one cannot be sloppy. Proof rules are not needed for operational proofs.

This chapter outlines our approach, which is based on Hoare logic and temporal logic. We describe the use of predicates to express properties of states, Hoare-triples to express properties of program statements and events, and temporal logic assertions to express properties of executions. We present proof rules for inferring Hoare triples and assertions. We use a subset of temporal logic, namely, “invariant” and “unless” assertions for safety properties, and “leads-to” assertions for progress properties. We allow so-called “auxiliary variables” in analysis; these are fictitious variables that record information about the system’s past behavior without influencing its execution. The definitions and theorems developed here apply to systems with deterministic or nondeterministic events.

5.2 Predicates

We use predicates for expressing properties of system states. A predicate is a boolean-valued function in variables. More precisely, it is a statement in first-order logic, with boolean-valued terms, logical connectives \neg (not), \wedge (and), \vee (or), \Rightarrow (implies), and quantifiers $\forall x$ (for all x), $\exists x$ (for some x), and $\exists! x$ (for exactly one x) where x is a variable. By convention, \neg , \wedge , \vee , \Rightarrow , $\forall x$, $\exists x$, and $\exists! x$ are in decreasing order of binding power. The terms in a predicate can involve non-boolean expressions and operators. The following is an example predicate, where x , y , and u are integer variables, f is a function from integers to integers, p is a process, and S is a statement.

$$x > y + z \vee p \text{ at } S \Rightarrow [\forall \text{int } u :: f(u) = f(y + u)]$$

The general form of a “ \forall ” predicate is $[\forall V :: Q]$, where Q is a predicate and V introduces **bound variables** along with their domains. It can be read as “for all values of variables in V , Q is satisfied”. The square brackets indicate the scope of the bound variables and the quantification. Variables that are not bound are **free variables**. In the predicate example above, z is bound and the others are free.

Similarly, the general form of a “ \exists ” predicate is $[\exists V :: Q]$. It can be read as “for some value of variables in V , Q is satisfied”. The general form of a “ $\exists!$ ” predicate is $[\exists! V :: Q]$. It can be read as “for exactly one value of variables in V , Q is satisfied”.

A system state s satisfies a predicate P iff P is true for the value assignment specified by s ; that is, P is true after every *free* variable of P that is assigned a value in s is replaced by its value in s . (Bound variables in P with the same name as variables in s may need to be renamed in order to avoid name clashes.) If P has

free variables F that are not assigned values in s , then s satisfies P iff s satisfies P for every value assignment of the variables in F , i.e., iff s satisfies $[\forall F :: P]$.

We need some additional notation for reasoning about systems with a dynamic set of processes and threads. First, every process/thread is assigned a unique (never before used) “abstract id” at creation time. This id is for analysis purposes, and is independent of any process id that may be implemented. Second, for any system q , define the variable $q.\pi\text{set}$ to be the set containing a $\langle p, P \rangle$ pair for each active process, where p is the abstract id of the process or thread and P is the name of the program or function that p started executing. Variable $q.\pi\text{set}$ is updated whenever a process or thread is started or terminated in q .

We also need some notation for reasoning about the control state of a process. For a process p and a statement S , the expression $p \text{ at } S$ is true iff control of p is at the start of S , and the expression $p \text{ in } S$ is true iff control of p is inside S . Note that $p \text{ in } S$ is always false for atomic S . The expression $p \text{ on } S$ is short for $p \text{ at } S \vee p \text{ in } S$. The expression $p \text{ at } S_1, \dots, S_n$ is short for $p \text{ at } S_1 \vee \dots \vee p \text{ at } S_n$. Expressions $p \text{ in } S_1, \dots, S_n$ and $p \text{ on } S_1, \dots, S_n$ are analogously defined. The following is an example predicate, where p is an abstract process id, S is a statement, and q is a system.

$$p \text{ at } S \Rightarrow |q.\pi\text{set}| = 1 \wedge p \text{ in } q.\pi\text{set}$$

5.3 Hoare-triples

We use Hoare-triples for expressing properties of program statements and events. Hoare-triples were originally introduced for expressing properties of sequential programs. In this context, for a statement S and predicates P and Q , the Hoare-triple $\{P\}S\{Q\}$ means that the execution of S starting from any state satisfying P always terminates in a state that satisfies Q (without encountering an undefined operation).

Here are some examples of Hoare-triples; with each we indicate whether or not it is valid:

- $\{\text{true}\} \text{ if } x \neq y \text{ then } x := y + 1 \{x = y + 1 \vee x = y\}$ (valid)
- $\{\text{true}\} \text{ await } x \neq y \text{ do } x := y + 1 \{x = y + 1\}$ (valid)
- $\{x = n\} x := 1 \square x := x + 1 \{x = 1 \vee x = n + 1\}$ (valid, where \square is nondeterministic selection)
- $\{x = n\} \text{ cobegin } \langle x := 1 \rangle \parallel \langle x := x + 1 \rangle \text{ coend } \{x = 1 \vee x = 2 \vee x = n + 1\}$
(valid, where $x := 1$ and $x := x + 1$ are atomic)
- $\{x = n\} \text{ for } i := 0 \text{ to } 10 \text{ do } x := x + i \{x = n + 55\}$ (valid)
- $\{x = 3\} x := y + 1 \{x = 4\}$ (invalid)
- $\{x = 1 \wedge y = 1\} \text{ while } x > 0 \text{ do } x := 2 * x \{y = 1\}$ (invalid – does not terminate)
- $\{\text{true}\} \text{ await } x \geq 1 \text{ do } y := 1/(2 - x) \{y = 1/(2 - x)\}$ (invalid – may divide by zero)

We extend Hoare-triples to concurrent systems as follows. For statement S , and predicates P and Q , $\{P\}S\{Q\}$ means that for every state s satisfying P , the *atomic* execution of S starting from s does not encounter any undefined operation and either terminates in a state that satisfies Q or blocks without terminating. If S has a function or event call that returns a value, then this must hold for *every possible return value*. If S has no blocking constructs, $\{P\}S\{Q\}$ means that the execution of S starting from any state satisfying P always terminates in a state that satisfies Q .

We extend the Hoare-triple notation to an event e with enabling condition and action by defining $\{P\}e\{Q\}$ to mean $\{P \wedge e.ec\}e.ac\{Q\}$. That is, for every state s satisfying P , either (1) e is not enabled at s , or (2) the execution of $e.ac$ starting from s terminates in a state that satisfies Q (recall that an event action has no blocking constructs). For example, $\{\text{true}\}e\{\text{true}\}$ states that $e.ac$ always terminates when executed starting from any state satisfying $e.ec$.

5.4 Safety assertions

We have two kinds of assertions for expressing safety properties of executions, namely, “invariant assertions” and “unless assertions”. These assertions impose conditions on the inter-event states of executions, not on intra-event states (recall the comment on observability in section 4.4).

An **invariant assertion** has the form $\Box P$, where P is a predicate. $\Box P$ (read “invariant P ”) means that P always holds. Formally, $\Box P$ holds for an execution iff the execution is fault-free and every inter-event state in the execution satisfies P . $\Box P$ holds for a system iff it holds for every fault-free execution of the system and the system has no locally-caused faulty executions.

An **unless assertion** has the form $P \text{ unless } Q$, where P and Q are predicates. It means that if P holds at some instant, then it continues to hold until Q holds. Formally, $P \text{ unless } Q$ holds for an execution iff the execution is fault-free and for every inter-event state in the execution that satisfies $P \wedge \neg Q$, either that state is the last state in the execution or the next inter-event state satisfies $P \vee Q$. $P \text{ unless } Q$ holds for a system iff it holds for every fault-free execution of the system and the system has no locally-caused faulty executions.

There is some trivial overlap between predicates, invariant assertions, and unless assertions. In particular, if a predicate P is always true (e.g., $2 > 1$, $[\forall \text{int } n :: n^2 > n]$), then $\Box P$ holds. Also, if $\Box(P \Rightarrow Q)$ holds then $P \text{ unless } Q$ holds.

5.5 Progress assertions

We have two kinds of assertions for expressing progress properties of executions, namely, simple “leads-to” assertions and compound “leads-to” assertions. Like invariant and unless assertions, leads-to assertions do not state conditions on intra-event states.

A **simple leads-to assertion** has the form $P \rightsquigarrow Q$, where P and Q are predicates. $P \rightsquigarrow Q$ means that if P holds at some instant, then Q holds at that instant or at some later instant. Formally, $P \rightsquigarrow Q$ holds for an execution iff the execution is fault-free and for every inter-event state in the execution that satisfies P , either that state satisfies Q or some later inter-event state satisfies Q . $P \rightsquigarrow Q$ holds for a system iff it holds for every *complete* execution of the system (i.e., execution that satisfies the system’s fairness assumptions).

A **compound leads-to assertion** is a predicate with its terms replaced by leads-to assertions, for example, $[\forall \text{int } n :: ((X \rightsquigarrow Y) \Rightarrow (P \rightsquigarrow Q))]$. A compound leads-to assertion R holds for an execution iff the execution is fault-free and R evaluates to **true** after each simple leads-to assertion S in R is replaced by **true** or **false** depending on whether or not the execution satisfies S . R holds for a system iff it holds for every complete execution of the system. [We do not allow the underlying predicate to have \neg ’s. This is to avoid assertions like $\neg(P \rightsquigarrow Q)$, which are really assertions about absence of progress.]

There is some trivial overlap between safety and progress assertions. For example, if $\Box(P \Rightarrow Q)$ holds then $P \rightsquigarrow Q$ holds.

By convention the temporal connectives (\Box , **unless**, \rightsquigarrow) bind weaker than logical connectives (\neg , \wedge , \vee , \Rightarrow) and stronger than quantifiers (\forall , \exists , $\exists!$). Thus $[\forall n :: P \rightsquigarrow Q \Rightarrow R]$ is parsed as $[\forall n :: (P \rightsquigarrow (Q \Rightarrow R))]$.

5.6 Increasing the granularity of atomicity

When analyzing a system, we are ultimately interested only in the external behavior of the system, that is, in the sequences of input and output calls it can display. We are interested in a system’s internal behavior only in so far as it is relevant to understanding the external behavior. This is why intra-event states are not used in evaluating safety and progress assertions. More precisely, because the intra-event states of an event execution are not observed or affected by the event’s environment, they do not affect the external behavior of the event, let alone the external behavior of the system.

In the same way, a sequence of events can be treated as atomic if the intermediate *inter-event* states in the sequence are not observed or influenced by the sequence’s environment, and hence do not affect the external behavior of the system. Such an event sequence is said to be **effectively atomic**. Given a system program, there is no easy way to identify all the effectively-atomic event sequences in the program. However, there are some simple sufficient conditions, and we next explore some based on statements that do not share variables.

Two statements S and R have **write-overlap** if they are executed by different processes and one statement writes a variable accessed by the other statement (including enabling the other statement). A statement S has a write-overlap with its environment if it has a write-overlap with a statement executed by the environment (i.e., an xc event action). Note that write-overlap is a syntactic condition, hence easily checked.

A finite sequence of atomic statements \bar{S} executed by a process is **blockable only at the start** if once the first statement in \bar{S} executes, the execution of the rest of \bar{S} cannot be blocked; that is, if $\bar{S} = \langle S_1, S_2, \dots, S_n \rangle$ then $\{S_{i-1}.ec\}S_{i-1}.ac\{S_i.ec\}$ and $\{S_i.ec\}f\{S_i.ec\}$ hold for $i = 2, \dots, n$ and any event f of another process.

The following theorems will be proved later.

Theorem 5.1 Let \bar{S} be a finite sequence of atomic statements executed by a process. Let \bar{S} be blockable only at the start. \bar{S} is effectively atomic if at most one atomic statement in \bar{S} has write-overlap with its environment. ■

Theorem 5.2 Let \bar{S} be a finite sequence of atomic statements executed by a process. Let \bar{S} be blockable only at the start. \bar{S} is effectively atomic if \bar{S} is mutually exclusive with every statement R that has a write-overlap with S (that is, $\Box \neg(S.ec \wedge R.ec)$ holds for every statement S in \bar{S} other than the first statement of \bar{S}). ■

Note that theorem 5.2 holds vacuously if \bar{S} consists of only one atomic statement. If \bar{S} has more than one statement, the program has to ensure that \bar{S} and R are mutually exclusive by using some synchronization mechanism. For example, consider a system program that has two processes, one repeatedly executing $\{P(\text{mutex}); S; V(\text{mutex})\}$ and the other repeatedly executing $\{P(\text{mutex}); T; V(\text{mutex})\}$, where mutex is initialized to 1. If the system program's xc events have no write-overlap with S and T , then S and T are effectively atomic, even if they have write-overlap and do outputs.

The difference between a statement being atomic and a statement being effectively atomic is that in the former case, the statement's atomicity is provided by the underlying platform irrespective of the system program in which the statement appears, whereas in the latter case, the statement's atomicity is provided by the system program in which the statement appears and is specific to that program.

5.7 Atomicity and commuting events

Let \bar{S} be a finite sequence of atomic statements executed by a process. Let \bar{S} be blockable only at the start. We establish a sufficient condition for \bar{S} to be effectively atomic, one that implies theorems 5.1 and 5.2. We start by introducing the notion of “commuting” events, which is fundamental to the notions of atomicity and serializability.

Given events g and f , we say $\langle g, f \rangle$ **commutes** if for every execution x that has an execution of g followed immediately by an execution of f , the order of the two executions can be reversed and the intermediate inter-event state modified so that the resulting sequence, say x' , is also an execution of the system. That is, if $x = \langle \bar{\alpha}, g, r, f, \bar{\beta} \rangle$, then there is a state s such that $x' = \langle \bar{\alpha}, f, s, g, \bar{\beta} \rangle$ is also an execution. Given a function F on executions (e.g., the external trace), we say $\langle g, f \rangle$ **commutes wrt** F if $F(x)$ equals $F(x')$.

Let \bar{g} be a finite sequence of events and g be an event of \bar{g} that is not the last. We say g is **tail-droppable** if for every execution $x = \langle \bar{\alpha}, g, \bar{\beta} \rangle$ such that $\bar{\beta}$ has no events of \bar{g} , there is a $\bar{\beta}'$ such that $x' = \langle \bar{\alpha}, \bar{\beta}' \rangle$ is also an execution. We say g is **tail-droppable wrt** a function F if $F(x)$ equals $F(x')$.

Let \bar{g} be a finite sequence of events and g be an event of \bar{g} that is not the first. We say g is **tail-appendable** if for every execution $x = \langle \bar{\alpha}, g', \bar{\beta} \rangle$ such that g' is the predecessor of g in \bar{g} and $\bar{\beta}$ has no events of \bar{g} , there is a $\bar{\beta}'$ and state s such that $x' = \langle \bar{\alpha}, g', s, g, \bar{\beta}' \rangle$ is also an execution. We say g is **tail-appendable wrt** a function F if $F(x)$ equals $F(x')$.

Theorem 5.3 Let \bar{S} be a finite sequence of atomic statements executed by a process. Let \bar{S} be blockable only at the start. Let the anchor of \bar{S} be an atomic event of \bar{S} . Let the following hold for any non-anchor event g of \bar{S} and any event f of a process other than the one executing \bar{S} :

- If g precedes the anchor, then $\langle g, f \rangle$ commutes wrt external behavior.
- If g succeeds the anchor, then $\langle f, g \rangle$ commutes wrt external behavior.
- If g precedes the anchor, then g is tail-droppable wrt external behavior.

- d. If g succeeds the anchor, then g is tail-appendable wrt external behavior.

Then \bar{S} is effectively atomic. ■

Proof Let x be an execution of the system. Because \bar{S} is executed by a process, the sequence of events of \bar{S} in x consists of zero or more instances of \bar{S} followed by a proper (perhaps empty) prefix \bar{S}' of \bar{S} .

Consider any instance of \bar{S} . The events in x corresponding to this instance of \bar{S} need not be contiguous in x . However, condition a allows us move forward the pre-anchor events in x corresponding to this instance until they are contiguous with the anchor of this instance (that is, repeatedly commute each pre-anchor event g in x with the event in front of it in x until g arrives immediately before its succeeding event in \bar{S} (which may be the anchor)). Similarly, condition b allows us move backward the post-anchor events in x corresponding to this instance until they are contiguous with the anchor of this instance.

So we obtain an execution, say y , in which the events of each \bar{S} instance are contiguous and the external behavior is the same as for x .

Now consider the events in y corresponding to the prefix \bar{S}' . If \bar{S}' does not include the anchor, then condition c tells us that we can drop these events from y and still have an execution, say z , that has the same external behavior as y (and hence x). If \bar{S}' does include the anchor, then condition d tells us that we can add the remaining events of \bar{S} to y and still have an execution, say z , that has the same external behavior as y (and hence x).

So the nonnull proper prefix \bar{S}' of \bar{S} is either removed or topped up to a complete \bar{S} instance. So we obtain an execution z that has the same external behavior as x and whose \bar{S} events form zero or more complete instances of \bar{S} . Thus z is an execution of the system obtained from the original system by treating \bar{S} as atomic. ■

Proof of theorem 5.1 Let \bar{S} satisfies the requirements of theorem 5.1. It suffices to show that \bar{S} satisfies conditions a-d of theorem 5.3. We first define the anchor of \bar{S} as follows: if \bar{S} has an event that has write-overlap with its environment, let that event be the anchor of \bar{S} ; otherwise let the anchor be any event of \bar{S} . Then \bar{S} satisfies conditions a-d because every non-anchor event g of \bar{S} has no write-overlap with its environment (work out the details). This establishes theorem 5.1.

Proof of theorem 5.2 Let \bar{S} satisfies the requirements of theorem 5.2. It suffices to show that \bar{S} satisfies conditions a-d of theorem 5.3. Define the anchor of \bar{S} to be the first event of \bar{S} . Then \bar{S} satisfies conditions a-d because every non-anchor event g of \bar{S} is mutually exclusive with any event with which it has write-overlap (work out the details). This establishes theorem 5.2.

5.8 Composite and component systems

Theorem 5.4 Let C be a composite system, let Q be a component of C , and let Y be a safety or progress assertion whose free variables have an empty intersection with the variables and events of components other than Q . Let x be a fault-free execution of C . Then x satisfies Y iff $x.Q$ satisfies Y . ■

Proof Because x is fault-free, $x.Q$ is a fault-free execution of Q (from Theorem 4.1).

We first consider Y that is an invariant, unless, or simple leads-to assertion. Thus Y holds for x iff Y holds for the sequence of states of x . Note that for any predicate whose free variables have no variables or events in common with components other than Q , the predicate holds for a state s of C iff it holds for $s.Q$. Thus Y holds for x iff Y holds for the sequence of Q images of states of x . Furthermore, a consecutive sequence of identical images can be replaced by a single occurrence of the image without affecting this equivalence. Thus Y holds for x iff Y holds for $x.Q$.

Now consider a compound leads-to assertion Y . Because Y is built up of assertions of the types considered above, the result holds for Y also. ■

5.9 Auxiliary variables

We sometimes find it convenient in the analysis of a system to introduce *fictitious* variables to record information about the system's past behavior without influencing its execution. These variables, called **auxiliary variables**, are only for analysis purposes; *they are not implemented*. They must satisfy the following **auxiliary variable condition**, which ensure that they do not affect system execution:

- Auxiliary variables do not appear in event enabling conditions, and their value is not used in updating a non-auxiliary variable. (A simple way to ensure this is to have auxiliary variables appear only in assignment statements, and if they appear in the right side of an assignment statement then the left side must be an auxiliary variable.)
- Statements involving auxiliary variables are well-formed. (That is, for every value assignment of the statement's variables satisfying type constraints, the execution of the statement is defined and terminates.)

Let Q be the system resulting by extending a system P with auxiliary variables. Then the executions of Q differ from those of P only in that each state of Q consists of a state of P and a state of the auxiliary variables. Thus the definition of image states and image executions is applicable here.

Theorem 5.5 Let Q be a system P extended with auxiliary variables.

- a. For every (fault or fault-free) execution x of Q , $x.P$ is an execution of P .
- b. For any execution y of P , there exists an execution x of Q such that $x.P$ equals y .
- c. For any assertion Y whose free variables have no auxiliary variables, P satisfies Y iff Q satisfies Y .

■

Proof

Part a. For any execution x of Q , $x.P$ is obtained by removing the auxiliary variable values from the states of x and collapsing consecutive intra-event states that differ only in auxiliary variable values. Because the auxiliary variables do not give rise to a fault and do not affect event enabling conditions or non-auxiliary variables, for every triple $\langle s_i, \bar{p}_i, s_{i+1} \rangle$ in x , $\langle s_i.P, \bar{p}_i.P, s_{i+1}.P \rangle$ is a transition of P . Hence $x.P$ is an execution of P .

Part b. Because the auxiliary variables do not give rise to a fault and do not affect event enabling conditions or non-auxiliary variables, for each transition $\langle s_i, \bar{p}_i, s_{i+1} \rangle$ in y , for every state t_i of Q such that $t_i.P = s_i$, there is a transition $\langle t_i, \bar{q}_i, t_{i+1} \rangle$ in Q such that $\bar{q}_i.P = \bar{p}_i$, and $t_{i+1}.P = s_{i+1}$. Thus there is an execution x of Q such that $x.P = y$.

Part c. If Q does not satisfy Y then there exists an execution x of Q such that x does not satisfy Y . From part a, $x.P$ is an execution of P . Because Y does not involve auxiliary variables, $x.P$ also does not satisfy Y . Hence P does not satisfy Y .

If P does not satisfy Y then there exists an execution y of P such that y does not satisfy Y . From part b, there exists an execution x of Q such that $x.P = y$. Because Y does not involve auxiliary variables, x does not satisfy Y . Hence Q does not satisfy Y .

■

5.10 Proof rules

So far we have described the syntax and semantics of assertions. We now describe some proof rules for assertions, starting with Hoare-triples, then safety assertions, and finally progress assertions. A proof rule consists of a list of conditions and a conclusion such that if the former hold then we can infer the latter. The conditions and conclusion are stated in terms of constructs of programs and assertions, and so their applications can, in principle, be mechanically checked.

Proof rules for Hoare-triples

A complete proof system for Hoare-triples would have a proof rule for every possible program construct and data structure. Here are proof rules for some program constructs:

- **Assignment rule:** $\{P\} x := t \{Q\}$ holds if $P \Rightarrow Q[x/t]$ holds, where $Q[x/t]$ is Q with every free occurrence of x replaced by t .
- **If-then-else rule:** $\{P\}$ if B then S else $T \{Q\}$ holds if $\{P \wedge B\}S\{Q\}$ and $\{P \wedge \neg B\}T\{Q\}$ hold.
- **Await rule:** $\{P\}$ await B do $S \{Q\}$ holds if $\{P \wedge B\}S\{Q\}$ holds.
- **While rule:** $\{P\}$ while B do $S \{Q\}$ holds if, for some nonnegative function T , $\{P \wedge B \wedge T = n\}S\{P \wedge T < n\}$ and $P \wedge \neg B \Rightarrow Q$ hold.
- **Sequential composition rule:** $\{P\}S; T\{Q\}$ holds if, for some predicate R , $\{P\}S\{R\}$ and $\{R\}T\{Q\}$ hold.

Proof rules for safety assertions

Here are proof rules for inferring safety assertions from system programs.

- **Invariance rule:** $\Box P$ holds for a system if the following hold:
 - (i) initial condition $\Rightarrow P$
 - (ii) for every event e : $\{P\}e\{P\}$
- **Unless rule:** P unless Q holds for a system if, for every event e , $\{P \wedge \neg Q\}e\{P \vee Q\}$ holds.

The soundness of these rules is straightforward. Consider the invariance rule. Consider any execution. From condition i, the initial state of the execution satisfies P . From condition ii and induction on the transitions, every transition is fault-free and every inter-event state satisfies P .

Note that each rule above imposes a condition for every event of the system program. In fact, if a sequence of events is effectively atomic, we can treat that event sequence as an event for the proof rule application. So in the above rules, we can replace “event” by “effectively-atomic event sequence”.

There are also proof rules for inferring assertions from other assertions, and we refer to these as **closure rules**. Here are some examples:

- $\Box P$ holds if P holds.
- $\Box P$ holds if $\Box Q$ and $\Box Q \Rightarrow P$ hold.
- P unless Q holds if $\Box P \Rightarrow Q$ holds.
- P unless Q holds if R unless S , $\Box P \Rightarrow R$, and $\Box S \Rightarrow Q$ hold.

Such rules are straightforward, and we refer to them as **closure rules**. rather than distinguishing each with a name.

Proof rules for progress assertions

Here are proof rules for establishing leads-to assertions from system programs (prove their soundness):

- **Weak-fair rule:** $P \rightsquigarrow Q$ holds for a system if the following hold for an event e subject to $W_{\text{fair}}(e)$:
 - (i) $(P \wedge \neg Q) \Rightarrow e.ec$
 - (ii) $\{P \wedge \neg Q\}e\{Q\}$
 - (iii) for every other event f : $\{P \wedge \neg Q\}f\{P \vee Q\}$

- **Strong-fair rule:** $P \rightsquigarrow Q$ holds for a system if the following hold for an event e subject to $S_{\text{fair}}(e)$:
 - (i) $(P \wedge \neg Q \wedge \neg e.ec) \rightsquigarrow (Q \vee e.ec)$
 - (ii) $\{P \wedge \neg Q\}e\{Q\}$
 - (iii) for every other event f : $\{P \wedge \neg Q\}f\{P \vee Q\}$

As in the case of the invariance and unless rules, we can replace “event” by “effectively-atomic event sequence” in the above rules.

Here are some closure proof rules for inferring progress assertions from other assertions:

- $P \rightsquigarrow Q_1 \vee Q_2$ holds if $P \rightsquigarrow P_1 \vee Q_2$ and $P_1 \rightsquigarrow Q_1$ hold.
- $P \rightsquigarrow Q$ holds if $(P \wedge I) \rightsquigarrow (I \Rightarrow Q)$ and $\Box I$ hold.
- $P_1 \wedge P_2 \rightsquigarrow Q_2$ holds if $P_1 \rightsquigarrow Q_1$, P_2 unless Q_2 , and $Q_1 \Rightarrow \neg P_2$ hold.
- $P \rightsquigarrow Q$ holds if P unless Q , $\Box P \Rightarrow R$, $R \rightsquigarrow S$, and $\Box S \Rightarrow \neg P$ hold.

Closure rules are straightforward and we do not give them distinct names, except for the following:

- **Well-founded leads-to rule:** $P \rightsquigarrow Q$ holds if, for some function F on a lower-bounded partial order $(Z, <)$, the following hold, where x and w range over Z :
 - (i) $P \rightsquigarrow Q \vee [\exists x :: F(x)]$
 - (ii) $F(w) \rightsquigarrow Q \vee [\exists x :: x < w \wedge F(x)]$

5.11 Concluding Remarks

The proof system described here is a subset of linear temporal logic. We have only three temporal operators (“invariant”, “unless”, and “leads-to”) and we allow them to be applied only to predicates. We are able to state and prove any correctness property because we allow auxiliary variables. To achieve completeness without auxiliary variables, one would need a full temporal logic [38], one that has more temporal operators (e.g., “next”, “until”, “past”), allows temporal operators to be applied to assertions, and has proof rules for negation of progress.

The use of auxiliary variables in program verification was introduced in [40]. A difference between our use of auxiliary variables and theirs is that we use them in stating and proving desired properties, whereas they used them only in proving; their desired properties were stated without recourse to auxiliary variables. Auxiliary variables are also called history variables in the literature.

Although fairness assertions define progress properties, we have chosen not to include them in our class of progress assertions. We shall use fairness assertions only in system programs, to define the progress *expected* of the underlying platform in scheduling system processes. One could also use fairness assertions to define *desired* progress properties of systems, but we do not do so. There is no loss of generality in our approach because fairness assertions can be expressed by leads-to assertions with the aid of auxiliary variables. For example, $W_{\text{fair}}(e)$ for an event e is equivalent to $(x = n \wedge e.ec) \rightsquigarrow (x > n \vee \neg e.ec)$, where x is an auxiliary integer variable that is incremented in the action of an event e . Similarly, $S_{\text{fair}}(e)$ is equivalent to $(\neg e.ec \rightsquigarrow e.ec) \Rightarrow (x = n \rightsquigarrow x > n)$

“Being invariant” versus “satisfying invariance rule”: “ X is invariant”, or $\Box X$, means X holds at every state of every execution, i.e., at every reachable state. “ X satisfies invariance rule” means that X holds initially and, for any state (reachable or not) satisfying X and any event, executing the event in the state results in a state that satisfies X . So “ X satisfies invariance rule” implies “ X is invariant”, but the converse is not true. For example, in problem 1 of Exam 1b, S_1 satisfies the invariance rule, and $nb + nc \leq na$ is invariant but it does not satisfy the invariance rule.

5.12 Exercises

- 5.1 Concerning Theorem 5.1, show that \bar{S} is not effectively atomic if \bar{S} has two statements with shared variables.
- 5.2 Concerning Theorem 5.2, show that \bar{S} is not effectively atomic if there is a statement R that shares a variable with \bar{S} and is not mutually exclusive with \bar{S} .

