

Concurrent Systems and Services: Specification and Verification

A. Udaya Shankar
shankar@cs.umd.edu

August 30, 2004

Contents

1	Introduction	3
1.1	Concurrent systems	3
1.2	Pie in the sky	4
1.3	Executable services	6
1.4	Layered compositionality	7
1.5	Atomicity and fairness assumptions	8
1.6	Assertional reasoning	9
1.7	Systems	10
1.8	Services	10
1.9	Satisfying services	11
1.10	Testing for satisfaction	12
1.11	Background	13

Chapter 1

Introduction

1.1 Concurrent systems

A **concurrent system** is a collection of active entities that execute simultaneously and interact with each other during the course of their lifetimes. Concurrency is inherent in the real world, arising whenever multiple active entities cooperate to achieve a goal. Small variations in the speeds of the agents can result in widely differing executions. Consequently, a concurrent system has many possible executions, and its behavior is usually not reproducible. This is what makes a concurrent system so interesting to behold, and so infuriating to debug.

The objective in concurrent system design is to ensure that *every possible execution satisfies the desired properties while allowing as much variation in speeds as possible*, that is, freeing the individual entities to run at their own speeds. Achieving this for systems of realistic complexity requires a “compositional” methodology for design and implementation, that is, one in which the design and implementation of a concurrent system can be broken up into the design and implementation of component concurrent systems. And that is what this book is all about, a *practical* and *rigorous* compositional methodology for concurrent systems. Our focus is on systems in the computer world, but the methodology applies to discrete-event systems in general.

In the computer world, the term **process** is used to refer to an active entity. A process executes a **program**. The program defines variables and statements, and the process executes the statements one by one. The **state** of the process consists of the values of its variables (including a control variable indicating the next statement that the process is to execute). Some statements only affect the state of the executing process. Some statements also affect the state of other processes, for example, writing to variables readable by other processes, sending messages to other processes, doing remote procedure calls, freeing a process from a wait, and even starting or terminating processes. When one process changes the state of another process, we say that an **interaction** has occurred between the processes, with the first process doing an **output** and the second process doing an **input**.

For a concurrent system with a single process, an **execution** is a sequence of statement executions along with the intermediate states. Some of the statement executions correspond to inputs from or outputs to other processes in the environment. Figure 1.1 illustrates such an execution. For a concurrent system with several processes, an execution consists of executions of its individual processes “stitched together” at their interactions, that is, at corresponding input and output pairs. Figure 1.2 illustrates this for a system of two processes. Clearly, small changes in the speeds of the individual processes can result in different executions of the concurrent system. In Figure 1.2 for example, if process 2 executed faster, its first output would happen before process 1’s first output, resulting in a very different overall execution.

The properties we are interested in are so-called **correctness properties**. A correctness property is nothing but a condition about the sequence of states that a system goes through; for example, “variable x never exceeds 5”, “statement Y is eventually executed after statement Z ”, and so on. A *correctness property is satisfied by a concurrent system if it is satisfied by every possible execution of the system*. Correctness properties can be classified into **safety** properties and **progress** (also called “liveness”) properties. A safety property states that “nothing bad” happens, for example, “variable x never exceeds 5”, “statement Y is

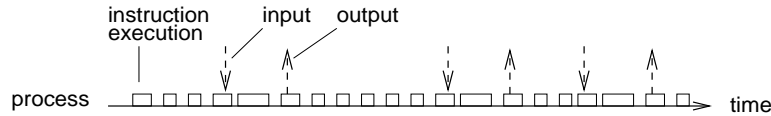


Figure 1.1: A possible execution of a single process concurrent system.

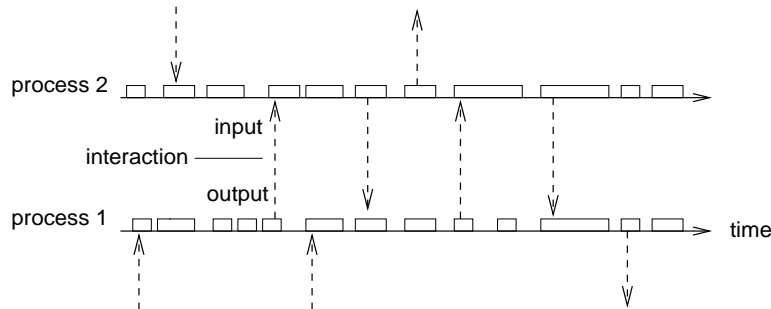


Figure 1.2: A possible execution of a concurrent system with two processes.

executed only after statement Z ", and so on. A progress property states that “something good” eventually happens, for example, “variable x eventually becomes 5”, “statement Y is eventually executed after statement Z ”, and so on. We use **assertions** (i.e., temporal logic formulas) to express correctness properties.

Concurrent systems are everywhere in the computer world, in the hardware level of processors and IO devices, in the system level of operating systems and network protocols, and in the application level of distributed applications and interactive user interfaces. They go by various names, including “distributed” systems and “interactive” systems, depending on the context. In particular, a distributed system is a concurrent system whose processes are spread over different locations and have only “coarse-grained” interactions with each other. An interactive system typically has a human user as one of the processes.

The non-computer world also is full of concurrent systems. A group of people interacting according to the rules of a game or a contract forms a concurrent system. An instructive example with close analogies to the computer world is that of several cooks in one kitchen. See Figure 1.3. Each cook (process) executes a recipe (program), which is made up of steps (statements). The cooks execute concurrently over time, each at his or her own speed. Each step of the recipe requires resources such as utensil, stove, and counter space. Because resources are shared there is interaction between the cooks, for instance, grabbing a pot only when it is free. A desired safety property is that the kitchen does not reach a state where all the pots are being used and each cook that has a pot needs another pot to complete his or her recipe (i.e., deadlock). A desired progress property is that every cook eventually gets enough pots to complete his or her recipe (i.e., no starvation).

1.2 Pie in the sky

Our goal is a **compositional methodology** for the design and implementation of concurrent systems, that is, one where the design and implementation of a concurrent system can be broken up into the design and implementation of component concurrent systems.

To achieve this, we need two descriptions of a system. One is a description of the system itself, that is, the program executed. The other is an easily understandable description of the desired *external behavior* of the system, capturing all desired safety and progress properties and unencumbered by implementation issues and internal structure. We refer to this description as the **service** of the system. The service defines the set of *all acceptable sequences of interactions between the system and its environment*.

To illustrate, consider a distributed lock manager system accessed by user processes at various locations. *Any* sequence of requests, acquires, and releases, is acceptable, provided at most one user has the lock at any time (safety) and no requesting user is denied the lock indefinitely (progress). But to achieve this service, the system internally has to implement many things, keeping track of which processes are requesting the lock,

proves (tests) whether a system satisfies its services by analyzing (executing) the composite program of system and services. Programs can be in any concurrent programming language, subject to some simple syntactic constraints.

- **Layered compositionality:** We consider a composite system to consist of layers of component systems, with services defining the acceptable sequences of interactions between layers. A service, rather than enclosing a system, is a “boundary” between systems in different layers, with the systems below “offering” the service and the systems above “using” the service.
- **Atomicity and fairness assumptions:** Our system programs explicitly state the atomicity and fairness expected of the underlying execution platform. Making these explicit eliminates a common source of ambiguity in concurrent programs, and is essential for any meaningful analysis.
- **Assertional reasoning:** We use assertional reasoning for analyzing programs, because it can achieve rigor without excessive formalism. Assertions are convenient way to codify the intermediate steps of an analysis. They can be proved by operational arguments or by proof rule applications, and they can be mechanically checked in testing.

The above features are discussed in further detail in the following sections. We emphasize that our approach handles concurrent systems and properties in their full generality. The concurrent system models are not restricted, for example, to finite state machines or Petri nets. The class of correctness properties we consider subsumes partial correctness, invariance, termination, deadlock-freedom, livelock-freedom, worst-case complexity, hard real-time properties, etc. This generality is essential because software systems, unlike hardware systems, have not proven amenable to finite state verification techniques.

1.3 Executable services

We want a methodology that is close to the programmer’s world. This leads us to use programs to specify services, resulting in so-called **executable services**. Although both systems and services are specified by programs, the two types of programs differ fundamentally because they have different purposes. A system program is intended for execution and hence must satisfy the computational, synchronization and other constraints of the underlying platform — for example, does it run in privileged mode, is it multi-threaded or multi-process, does it execute on a single processor, a multi-processor with shared memory, or loosely-coupled message-passing processors, and so on. A service program, on the other hand, is intended *not for execution* but to provide an easily understandable definition of the service. Hence it can ignore the constraints of the underlying platform, for example, resorting to system-wide updates and global history variables. *It is precisely this freedom from “physical-world” constraints that makes a service program easier to understand than a system program that provides the service. A clean programming language is verry helpful but it is not the key.*

Consider the service provided by a distributed lock manager. The service can be succinctly defined by a program that has (1) a variable indicating the set of users currently requesting the lock, (2) a variable indicating the user, if any, currently having the lock, and (3) for each operation (request, acquire, release), a predicate indicating whether the operation is allowed and a function that updates the variables appropriately. The program cannot be directly executed on the underlying distributed platform because the acquire operation involves variables at different locations.

Because services are executable, the notion of a system satisfying a service can be formalized in terms of the **composite program of the system and service** satisfying certain properties. Hence, given a system and service, any concurrent program verification technique can be used to check the composite program for the properties. The technique we use is assertional reasoning (outlined below).

Another benefit of executable services is that it provides a **testing harness**: one can test a system against its services simply by executing the composite program of the system and services (on a “serialized” version of the underlying platform). Such a testing harness can be an invaluable complement to manual verification, in which one often “invents” assertions without checking the details. These assertions can be mechanically checked during testing of the composite program of system and services. Of course, testing explores only some of the finite executions of the composite program, and hence is not a substitute for verification.

1.4 Layered compositionality

In traditional compositionality approaches, illustrated in Figure 1.4, a service encloses a system and defines the acceptable sequences of interactions *between the system and its environment*. When component systems are composed to form a composite system, the behavior, and hence the service, of the composite system is determined by composing the services of the component systems.

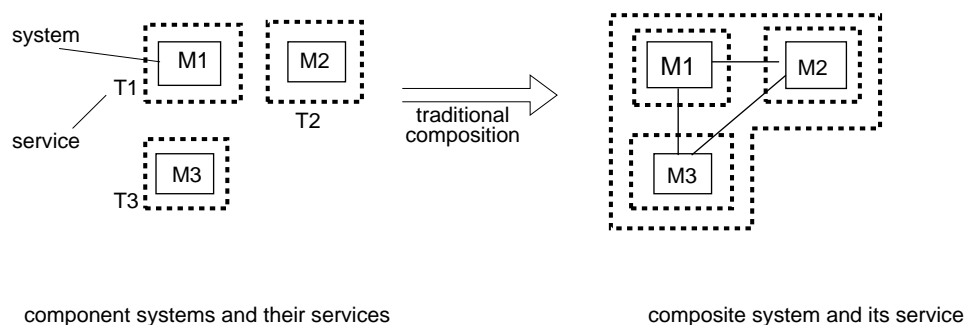


Figure 1.4: Traditional compositionality: service encloses a system.

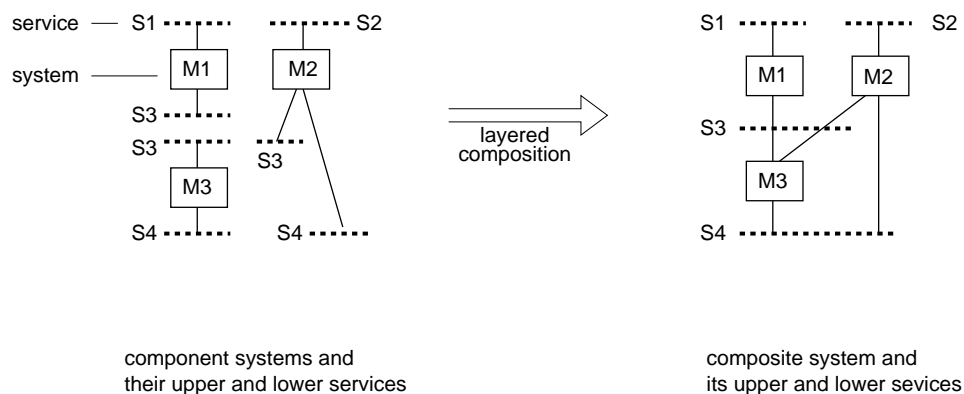


Figure 1.5: Layered compositionality: service is between layers.

In **layered compositionality**, illustrated in Figure 1.5, a composite system consists of layers of component systems, with services defining the acceptable sequences of interactions *between systems in different layers*. Thus a system, in general, is “encapsulated” by services above and below. When component systems are composed to form a composite system, services between components become internal to the composite system and the remaining services encapsulate the composite system. In Figure 1.5, M1 is encapsulated by S1 above and S3 below, while the composite system of M1, M2, M3 is encapsulated by S1 and S2 above and S4 below.

Encapsulating a system by services above and below is *not* equivalent to enclosing the system by a single service. For example, M1 in Figure 1.5 is encapsulated by S1 above and S3 below, but S1 and S3 together are not equivalent to the T1 enclosing M1 in Figure 1.4. Of course, if a system has only one service (either upper or lower), then that service encloses the system exactly as in traditional compositionality.

Roughly speaking, a system “satisfies” its encapsulating services if the interactions it initiates are allowed by the services, *assuming* the interactions initiated by the system’s environment are allowed by the services. Note that service satisfaction is a *conditional* obligation; the system is under no obligation if the environment does not meet *its* obligation. Given a system M and services U and V , we say M **satisfies U above and V below**, or as we prefer say, M **offers U uses V** , to mean that M is encapsulated by U above and V below and satisfies the services.

Our **compositionality property** is that, given a composite system consisting of layers of component systems with services in between, if every component system in isolation satisfies its services, then the

composite system as a whole satisfies its services. For example in Figure 1.5, if M1 offers S1 uses S3, M2 offers S2 uses S3, and M3 offers S3 uses S4, then the composite system of M1, M2, and M3 offers S1 and S2 uses S4.

When designing a composite system to offer a desired service, our approach is to first envision the intermediate services in the composite system, and then introduce component systems that satisfy the services above and below. The essence of layered compositionality is that services govern the interaction *between* layers but not the interaction within a layer. Of course, one can always introduce services between systems in the same layer, effectively placing these systems in different layers. Also, any two components can directly interact, even components separated by several layers. Also, a system that accesses a service can itself be made up of several systems. In fact, this is inevitable if the service is distributed over multiple locations, as illustrated in Figure 1.6. Thus, *layered compositionality does not constrain the kinds of composite systems that can be built*, unlike say the notion of layering in network protocol architecture.

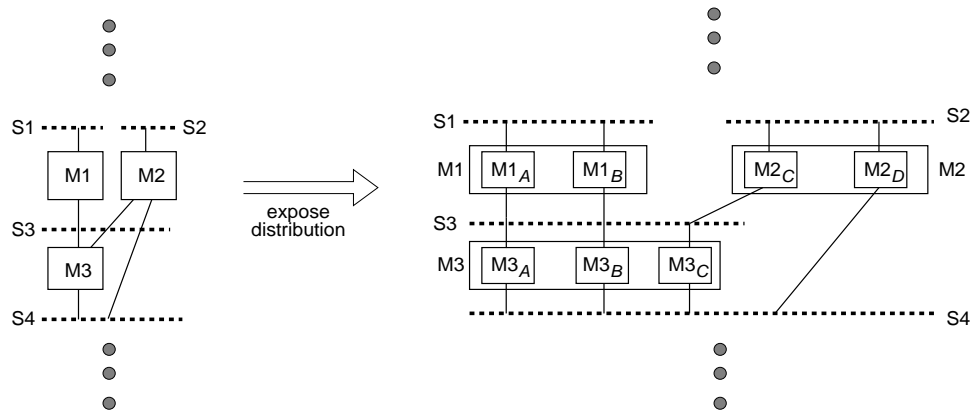


Figure 1.6: A layered composite system distributed over locations A, B, C, and D.

1.5 Atomicity and fairness assumptions

A concurrent program has constructs for inter-process communication (shared variables, locks, message sends and receives, etc.) and constructs for process creation, termination, and management (fork-join, cobegin-coend, setting scheduling priority, etc.). In order to say anything meaningful about what these constructs do, one has to look underneath and explicitly address the **atomicity** and the **fairness** expected of the underlying platform. Most computing platforms are ambiguous in this respect.

A concurrent program expects certain statements to be **atomically executed** by the underlying platform. Atomic execution of a statement means that *once the execution starts, no other simultaneously executing statement can influence the execution or observe intermediate states of the execution*. Specifying atomicity is equivalent to indicating the points at which the environment can interrupt or observe the program's execution. This atomicity may be as low-level as atomic reads and writes of memory words, say as provided by hardware, or as high-level as atomic transactions, say as provided by a database platform. In any case, the underlying platform must provide some level of atomicity. Without atomicity, the program's execution can be interrupted willy-nilly by its environment, for instance, in the middle of accessing a memory location, and *nothing* can be predicted about its behavior.

An atomic execution of a statement appears, to its environment, to occur *instantaneously* at some point between the start and the end of the execution. This allows one to use the **nondeterministic interleaving model** of execution, in which the simultaneous execution of atomic statements is represented by the set of all possible sequential executions of the atomic statements. Figure 1.7 illustrates this for two statements. Because the interleaving model permits the notion of a global state, it greatly facilitates reasoning about concurrent systems. *We emphasize that adopting the interleaving model does not mean that atomic executions must be non-overlapping in time.*

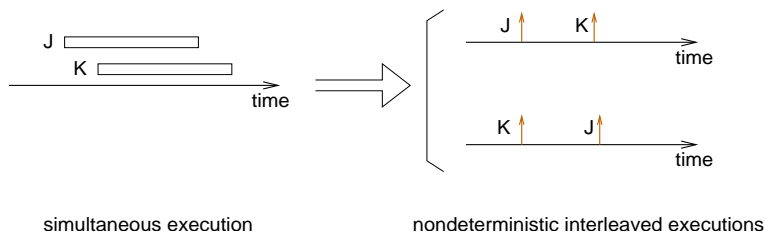


Figure 1.7: Concurrent execution modeled as nondeterministic interleaving.

A concurrent program also expects its processes to be executed with **fairness** by the underlying platform. There are two kinds of fairness: weak and strong. A process executed with **weak fairness** means that every once in a while the process gets some processor cycles. So if the process is at a nonblocking statement S , then it *eventually* executes that statement. What if the process is at a blocking statement, such as “await B do S ”, where the boolean B is affected by other processes. If B holds *continuously*, then the process eventually executes the statement. If B holds *infinitely often* but only intermittently, then it may never execute the statement (because B may happen to be false whenever the process gets cycles). However, if the process is executed with **strong fairness**, then even in the latter situation, it eventually executes the statement.

Clearly, it is not easy for a platform to provide strong fairness in general. Platforms usually provide only weak fairness, although even this is sometimes questionable. If at all a platform provide strong fairness, it is for a very limited set of blocking constructs (e.g., semaphore waits).

1.6 Assertional reasoning

Analysis ultimately boils down to showing that a given program satisfies a desired property, or, more realistically, a sequence of intermediate properties leading up to the desired property. We need a convenient formalism to express these properties, and our choice is to use **assertions** (temporal logic formulas). **Assertional reasoning** refers to the use of assertions to codify the intermediate results of analysis such that each result is easily derivable from previous results. In addition to conveniently expressing properties of programs, assertions can also be mechanically evaluated during testing.

We use three kinds of assertions: “invariant” and “unless” assertions for safety properties, and “leads-to” assertions for progress properties. An **invariant** assertion has the form $\square P$, where P is a predicate; it means that P holds at every possible state of the program. 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. A **leads-to** assertion has the form $P \rightsquigarrow Q$, where P and Q are predicates; it means that if P holds at some instant then eventually Q holds. One can also combine these assertions to form more complex assertions. For example, the desired progress property of the lock manager, i.e., every request is eventually followed by a corresponding acquire *assuming* that every user who has the lock eventually releases it, would be expressed by an assertion of the form $[(\text{acquire} \rightsquigarrow \text{release}) \Rightarrow (\text{request} \rightsquigarrow \text{acquire})]$.

Consider one step of an analysis, in which we want to prove that a desired assertion holds for a program assuming previously established assertions. There are two fundamentally different ways of expressing the proof: assertional and operational. In an **assertional proof**, one shows that the desired assertion follows by an application of a proof rule (i.e., temporal-logic inference rule). In an **operational proof**, one argues informally in natural language. Both have merits.

An assertional proof works with the syntactic constructs of the program and assertions. Consequently, a proof, once obtained, can be *checked* without understanding the program or the assertions. Indeed, it can even be mechanically checked with current theorem-proving technologies, but only if the proof is at a horribly low level of detail. This rarely happens, whether in real life or in this book.

An operational proof, on the other hand, works directly with the executions of the program rather than the syntax of the program. It can provides insight but it is error-prone. The accuracy and readability of an operational proof depends entirely on the analyst’s maturity and clarity of expression. People usually prefer operational proofs to assertional proofs because it is easier to skip details at the risk of being sloppy.

Overall, for the general class of systems and services of interest to us, proofs require human insight and invention. The greater the effort expended, the more trustworthy is the proof. Pragmatically, we recognize that the extent of precision and formality depends on factors such as how complex is the module, how well-understood are its algorithms, how expensive would a mistake be, and how much time and resources are available for verification.

1.7 Systems

The discussion so far has been on the “philosophy” underlying SESF. We now outline SESF itself, starting with systems, then services, and finally service satisfaction.

A system is specified by a **system program**. We use a generic concurrent programming syntax (along the lines of Java), with key words to identify the following: (1) the statements to be atomically executed, (2) the statements that are callable by the environment and their “enabling conditions” (described below), and (3) the fairness expected of the underlying platform.

In fact, a system program can be in any concurrent programming language (e.g., Java, C++/sockets, Ada, assembly). In this case, the above key words can be included in the program as comments; i.e., SESF becomes a markup language for program text. Such a SESF-structured Java is presented in [?].

The statements of a system program, as usual, can modify variables, call functions, create processes and start them executing, and so on. We use “:=” for assignment and “=” for equals. We use the term **events** to refer to atomically executed statements. An event is either **locally controlled** or **externally controlled**, depending on whether its execution is initiated by the system or by the environment. A locally-controlled event can call externally-controlled events of *other* systems. An externally-controlled event is executed only when called by a system in the environment. It does not call other events (but it can return a value to the caller). An externally-controlled event has an associated **enabling condition**, a predicate in the program variables that should hold whenever the event is called. **Fairness assumptions** define the fairness that is expected of the underlying platform in executing locally-controlled events.

Any collection of systems can be grouped to form a **composite** system. In addition to interactions with its environment, a composite system can also have internal interactions, that is, interactions between its components. Naturally, a component system can itself be a composite system.

A system is said to become **faulty** if it executes an undefined operation (division by zero, calling a non-existent event, etc.). For a properly designed system, this should happen only if an externally-controlled event is executed when its enabling condition does not hold. We do not care what a system does after it becomes faulty; it may crash, execute abnormally, make other systems faulty, and so on.

1.8 Services

A service is specified by a **service program**. The purpose of the program is to define in a convenient way (1) the signatures of the system events on each side that are callable by systems on the other side, and (2) the acceptable sequences of these event calls. Our approach to achieving this is to have the service program be a special kind of system program whose events are divided into **upward** events and **downward** events. An upward event corresponds to an externally-controlled event of a system above the service; it can be called only by a system below the service. A downward event corresponds to an externally-controlled event of a system below the service; it can be called only by a system above the service. Figure 1.8 illustrates this. Each event has an associated enabling condition. Each execution of the service program corresponds to an allowed sequence of event calls.

The program’s sole purpose is to express these sequences in a humanly understandable manner. The program is never implemented except for testing, and so efficiency and platform constraints are irrelevant. For example, the program of a distributed service can maintain a system-wide history that is updated by event executions at different locations.

The service program includes progress assertions that define the progress that is expected in executing upward events. They do not require any progress of downward events. The progress requirement is the only place where offering a service is not symmetric to using a service (if one classifies service events not

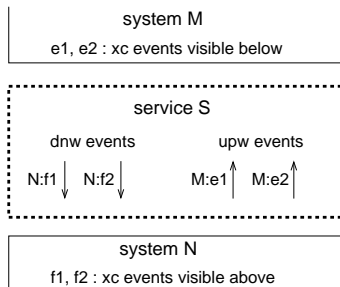


Figure 1.8: A service program between two systems (“upw”, “dnw”, and “xc” stand for “upward”, “downward”, and “externally-controlled”, respectively).

into upward and downward but into locally-controlled and externally-controlled with respect to the system accessing the service).

Most services in the concurrent world are so-called **multi-user** services, that is, services that support multiple users simultaneously. Internet IP service is an example of a multi-user service; here, the users are identified by IP addresses.

1.9 Satisfying services

What does it mean for a system M to satisfy a service U above. At any point in time, M has participated in a (possibly empty) sequence of U event calls, that is, calls of events corresponding to events of U . Of these event calls, the upward ones have been initiated by M and the downward ones by M 's environment. We say this sequence is **safe** if it is one of the sequences defined by U 's program. We say a U event call is **safe** at this point if extending the sequence by this call results in a safe sequence.

The notion of M **satisfying U above**, also said as M **offers U** , is formalized by the following:

- Safety condition: M does only safe U event calls, as long as M 's environment does only safe U event calls.
- Progress condition: M must initiate U event calls to satisfy the progress assertions of U , as long as M 's platform satisfies its fairness expectations and M 's environment does only safe U event calls.

Because the above conditions are stated in terms of the sequences of event calls in the executions of M and U , they are difficult to apply. It is preferable to have conditions stated in terms of the *programs* of M and U . This can be done simply by modifying M and U so that they interact directly. Let “ M -wrt- U ” be the system program obtained by redirecting every output of M to the corresponding event of U . Let “ U -wrt- M ” be the system program obtained from U by treating its upward events as externally-controlled events (to be called by M), and treating its downward events as locally-controlled events that call the corresponding (externally-controlled) events of M .

Let M^* denote the composite program consisting of M -wrt- U and U -wrt- M . The safety and progress conditions of M offers U can be restated, at the program level, as follows:

- Safety condition: M^* has only fault-free executions.
- Progress condition: M^* satisfies U 's progress assertions assuming M 's progress assertions.

The program-level reformulation has two crucial advantages: it can be checked with any program verification technique, and it can be mechanically tested by executing the composite program M^* . We emphasize that M^* is a hypothetical system, relevant only for analysis and testing.

The notion of a system using a service can be similarly formalized at the program level. Consider a system M that is encapsulated by a service V below. Let M^* denote the composite program consisting of M -wrt- V and V -wrt- M . M **satisfies V below**, also said as M **uses V** , if the following holds:

- Safety condition: M^* has only fault-free executions.

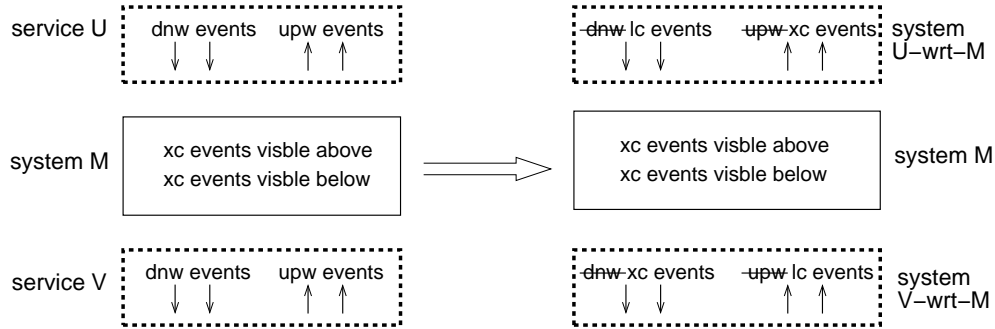


Figure 1.9: Composition of a system program and its service programs.

- Progress condition: None (because the progress assertions of V apply only to upward events).

The notion of a system offering a service and using a service is just the combination of the above. Consider a system M encapsulated by services U above and V below. Let M^* denote the composite program consisting of $M\text{-wrt-}\{U, V\}$, $U\text{-wrt-}M$, and $V\text{-wrt-}M$, illustrated in Figure 1.9. M **satisfies U above and V below**, also said as M **offers U uses V** , if the following hold:

- Safety condition: M^* has only fault-free executions.
- Progress condition: M^* satisfies U 's progress assertions assuming the progress assertions of M and V .

These definitions of satisfaction are compositional; that is, in a layered composite system, if every layer satisfies its services then the layered system satisfies its services.

It is straightforward to generalize this to the case of a system that satisfies multiple services above and below. It is also straightforward to generalize this to “partial-services”, which are services in which only some of the upward events are available for use (because the other upward events are used by the system offering the service). Partial-services typically arise with multi-user services in which some users of the service are actually part of the offering system.

1.10 Testing for satisfaction

The program-based formulation of service satisfaction gives us a way to mechanically test a system against services. Consider a system M that is encapsulated by services U above and V below. One can test for M offers U uses V by (1) constructing the composite system M^* of $M\text{-wrt-}\{U, V\}$, $U\text{-wrt-}M$ and $V\text{-wrt-}M$, and (2) executing M^* and checking whether the generated execution becomes faulty or does not satisfy the progress condition. Furthermore, the check can be expanded to include any assertion about the system that one would like to test.

Executing M^* is not straightforward because M^* has much higher atomicity requirements than M . Let I refer to the platform on which M is intended to execute. It is crucial in testing that the component $M\text{-wrt-}\{U, V\}$ of M^* also be executed on the platform I . Otherwise, one would have to modify M 's platform-dependent constructs, typically covering IO, communication, synchronization, and other concurrency-related capabilities. Such modifications would not only be very onerous, but they would most likely change M to a point that defeats the very purpose of testing.

On the other hand, platform I almost certainly cannot ensure atomicity of the interactions between $M\text{-wrt-}\{U, V\}$ and $U\text{-wrt-}M$ and between $M\text{-wrt-}\{U, V\}$ and $V\text{-wrt-}M$. This is because U and V , and hence $U\text{-wrt-}M$ and $V\text{-wrt-}M$, would, for any nontrivial service, make use of “unreasonable” atomicity.

One way to solve the problem is to implement on platform I a “serializer” system that interacts with $M\text{-wrt-}\{U, V\}$, $U\text{-wrt-}M$, and $V\text{-wrt-}M$ to “serialize” the execution of M^* . The serializer can also collect the global snapshots necessary to evaluate any assertion being checked.

1.11 Background

The key distinction between a concurrent system and a sequential system, for purposes of analysis, is that a concurrent system interacts with its environment during the course of its execution whereas a sequential system interacts with its environment only at the start and the end of its execution. So the state of an executing concurrent system can be changed at any time by other systems in the the environment, whereas the state of an executing sequential system can be changed only by itself. Handling this is what distinguishes the analysis of sequential systems from the analysis of concurrent systems.

The area of concurrent systems analysis has a rich history, and this section outlines where our approach stands with regard to modeling and analysis.

System models

Concurrent systems, whether they are realized in software or hardware, are almost always defined by programs in some programming language. But these programs themselves are rarely analyzed, invariably because the cost of analysis exceeds the rewards. Any analysis is done on concise *models* of the systems.

We have chosen to go with program models and assertional reasoning, because this combination can handle general concurrent systems and properties. A large body of experience suggests that this is the most effective foundation for understanding concurrent systems (for example, [14, 15, 16, 18, 19, 17, 21, 20, 23, 26, 29, 31, 34, 40, 41, 5, 46, 47, 49, 50, 51, 52, 53, 4]). Of course, analyses requires human insight and cannot be automated.

We use a generic programming syntax in this book, rather than a real-life language like Java. Our syntax has constructs for dynamic creation, naming, and deletion of objects, processes and threads. But beyond this, we avoid “ugly” programming language issues. Concurrent systems are hard enough to design without having to cater to the ambiguities and intrigues of programming languages. The SESF-Java work in [?] demonstrates that our approach can be used on real-life languages.

An alternative to program models is finite state machine models. These models can be automatically analyzed (e.g., [?]), but they suffer from two serious drawbacks. First, they cannot handle parametric properties; for example, a lock manager program for N users can be analyzed for a particular number of users, say 10, but not for N users. Second, even the most powerful finite-state analysis techniques cannot handle systems with practically unbounded variables, for example, integers, sequences, trees, graphs, and so on. It is no wonder that finite state approaches are very effective in digital hardware verification, but have not made significant headway outside that area.

Interleaving and partial-order semantics

Our assertional reasoning is founded on interleaving semantics. Here, the concurrent execution of events is modeled by nondeterministic interleaving of the events, resulting in a set of totally-ordered executions. The alternative to interleaving semantics is so-called *partial-order* semantics, where the concurrent execution of events is modeled by a single partial order. To illustrate, consider an event a being followed by events b and c , where b and c are unrelated and concurrently executed. Interleaving semantics would model this by the set $\{a \rightarrow b \rightarrow c, a \rightarrow c \rightarrow b\}$, whereas partial-order semantics would model this by the partial order $\{a \rightarrow b, a \rightarrow c\}$. Both semantics are equivalent in the sense that a property holds in one iff it holds in other. But they differ in other senses. The linear executions of interleaving semantics are easier for abstract reasoning, whereas partial orders are computationally more efficient for finite state machine analysis.

Assertional reasoning

Reasoning assertionally about programs is not new. Assertional reasoning for sequential programs was introduced by Floyd [22]. Hoare formalized this into a program logic, namely, Hoare-logic [24, 6], in which properties are expressed by triples of the form $\langle \text{precondition}, \text{code}, \text{postcondition} \rangle$ and proved by applications of inference rules. Dijkstra [14] introduced the concept of weakest preconditions and developed a program calculus for nondeterministic sequential programs with guarded commands.

Assertional reasoning for concurrent programs requires some extensions. First, because the processes of a concurrent program interact (and interfere) with each other, it is necessary to assume some level of atomicity

in their interaction [13]. Second, the properties of interest for concurrent programs are more complex than for sequential programs; we are interested, for example, in infinite executions where every nonblocked process eventually executes some statement [44]. Invariants and termination suffice for sequential programs, but not for concurrent programs. Pnueli [44, 45] pioneered the use of temporal logic for reasoning formally about the properties of concurrent systems. Since then, various assertional formalisms based on temporal logic have been proposed, for example, [1, 3, 37, 38, 42, 30, 32, 33, 11, 12, 7, 8, 35, 36, 46, 27].

We distinguish between *assertional reasoning* and *assertional proof*. Assertional reasoning refers to the use of assertions to codify intermediate results of an analysis. In addition to being convenient and unambiguous, assertions can also be mechanically checked during testing. Assertional proof refers to the use of proof rules to derive assertions. For every program construct there is a proof rule for inferring assertions that hold for that construct. Thus one can prove assertions about a program without ever reasoning *directly* about its executions. Assertional proofs are more trustworthy, but people usually prefer to do operational proofs because it is easier to skip details and be sloppy.

External behavior and compositionality

Once we have a method for expressing and proving properties of systems, the next step is to consider the issues of services, satisfying services, compositionality, and stepwise refinement. There are several formalizations based on assertional reasoning that address these issues, including [1, 3, 7, 8, 9, 11, 12, 30, 32, 33, 35, 28, 37, 48, 54]. All these formalizations share the basic notions of service, service satisfaction, and compositionality. That is, (1) the service of a system defines *all* (and only those) properties of the system that are of interest to its environment; (2) a system satisfies a service if the externally visible part of every execution of the system is allowed by the service; and (3) in a collection of systems, if each system satisfies a corresponding service, then the composition of the systems satisfies the composition of the services. However, the formalizations can differ in technical specifics.

One difference is whether the externally visible part of a system consists of event (function) calls or shared variables, that is, whether inputs are event calls or accesses (reads/writes) of variables. The two approaches are equivalent in expressive power, although they differ in technical development and, of course, syntax. The event call approach has become dominant in programming paradigms.

Another difference between the formalisms is whether or not services allows valid inputs that are blockable, for example, a wait operation. Some formalisms [25] allow this; that is, a system can have an input P such that when the environment makes a valid call of P , the system blocks the caller until some later time at which the call completes. Other formalisms [35, 8, 30, 27], including ours, do not allow this: a valid input cannot be blocked. In such as formalism, a blockable input P of a system M would be modeled by two events, say, $P.call$, an input of system M , and $P.return$, an input of the caller system. Again, both approaches are equivalent. We chose a nonblockable inputs because the theory is simpler; with blockable inputs, the external semantics of a system consists of a set of finite traces and associated refusal sets [25].

It turns out that achieving compositionality is not a straightforward matter. The service of a system, in general, consists of assumptions on the environment and requirements on the system, and the difficulty is in avoiding circular reasoning of progress properties. Each formalism imposes constraints in order for compositionality to hold. In our approach, circular reasoning is avoided by having service progress properties depending on the progress of only the “upward” events.

Stepwise refinement

Stepwise refinement is the converse to compositionality. We start with a service S and ask the question: how can we construct in an incremental fashion a system M that satisfies S . More precisely, is there a collection of “refinement” steps such that any system derived from S by applications of these steps results in a system that satisfies M . Thus compositionality is about growing outwards, whereas stepwise refinement is about growing inwards.

The most successful approach to stepwise refinement is to incrementally develop the system in the course of identifying additional properties, as demonstrated by Dijkstra in his numerous program derivations, e.g., [14, 15, 16]. In this approach, one starts with a skeleton system and a set of desired properties, and successively adds/modifies variables, events, and desired properties. The construction ends when we have a

system and a set of properties that satisfy the proof rules. There are several formalizations of this stepwise refinement approach, for example [1, 7, 9, 11, 12, 30, 32, 33, 35, 48].

Bibliography

- [1] M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 1991.
- [2] M. Abadi and L. Lamport. An Old-Fashioned Recipe for Real Time. Technical Report SRC-91, DEC SRC, Palo Alto, CA, October 1992. Earlier version in REX Workshop “Real-Time: Theory in Practice”, Netherlands, June 1991, LCNS 600.
- [3] M. Abadi and L. Lamport. Composing Specifications. *ACM TOPLAS*, 1993. Also in Stepwise Refinement of Distributed Systems, LNCS 430, Springer-Verlag, 1990.
- [4] C. Alaettinoglu and A.U. Shankar. Stepwise Assertional Design of Distance-Vector Routing Algorithms. In *IFIP Proceedings of 12th International Symposium on Protocol Specification, Testing, and Verification (PSTV)*, 1992.
- [5] G.R. Andrews. A Method for Solving Synchronization Problems. *Science of Computer Programming*, 1989.
- [6] K.R. Apt. Ten Years of Hoare’s Logic: A Survey-part I. *ACM TOPLAS*, 1981.
- [7] R.J.R. Back and R. Kurki-Suonio. Decentralization of Process Nets with a Centralized Control. In *Second ACM PODC (Principles of Distributed Computing)*, 1983.
- [8] R.J.R. Back and R. Kurki-Suonio. Distributed Cooperation with Action Systems. *ACM TOPLAS*, 1988.
- [9] R.J.R. Back and K. Sere. Stepwise Refinement of Parallel Algorithms. *Science of Computer Programming*, 1990.
- [10] A. Bernstein and P. Harter Jr. Proving Real-Time Properties of Programs with Temporal Logic. In *8th Symp. on Operating Systems Principles*, December 1981.
- [11] K.M. Chandy and J. Misra. An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection. *ACM TOPLAS*, 1986.
- [12] K.M. Chandy and J. Misra. *A Foundation of Parallel Program Design*. Addison-Wesley, 1988.
- [13] E.W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 1965.
- [14] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.
- [15] E.W. Dijkstra. A Correctness Proof for Communicating Processes – A Small Exercise. Technical report, Burroughs, Nuenen, The Netherlands, 1977. EWD-607.
- [16] E.W. Dijkstra. Two Starvation-free Solutions of a General Exclusion Problem. Technical report, Plataanstraat 5, 5671, AL Nuenen, The Netherlands, 1978. EWD-625.
- [17] E.W. Dijkstra, W.H.J. Feijn, and A.J.M. van Gasteren. Derivation of a Termination Detection Algorithm for Distributed Computations. *Information Processing Letters*, 1983. Also EWD 840.

- [18] E.W. Dijkstra, L. Lamport, A.J. Martin, and C.S. Scholten. On-the-fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, 1978.
- [19] E.W. Dijkstra and C.S. Scholten. Termination Detection for Diffusing Computations. *Information Processing Letters*, 1980.
- [20] N.J. Drost and A.A. Schoone. Assertional Verification of a Reset Algorithm. Technical report, Rijksuniversiteit Utrecht, 1988. RUU-CS-88-5.
- [21] N.J. Drost and J. van Leeuwen. Assertional Verification of a Majority Consensus Algorithm for Concurrency Control in Multiple Copy Databases. Technical report, Rijksuniversiteit, Utrecht, 1988. RUU-CS-88-13.
- [22] R.W. Floyd. Assigning Meanings to Programs. In *Proceedings of the Symposium on Applied Mathematics*, 1967.
- [23] B.T. Hailpern and S.S. Owicki. Modular Verification of Computer Communication Protocols. *IEEE Transactions on Communications*, 1983.
- [24] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 1969.
- [25] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.
- [26] D.E. Knuth. Verification of Link-level Protocols. *BIT*, 1981.
- [27] S.S. Lam and A.U. Shankar. A Relational Notation for State Transition Systems. *IEEE Transactions on Software Engineering*, 1990.
- [28] S.S. Lam and A.U. Shankar. Specifying Modules to Satisfy Interfaces: A State Transition System Approach. *Distributed Computing*, 1992.
- [29] L. Lamport. An Assertional Correctness Proof of A Distributed Algorithm. *Science of Computer Programming*, 1982.
- [30] L. Lamport. Specifying Concurrent Program Modules. *ACM Transactions on Programming Languages and Systems*, 1983.
- [31] L. Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 1987.
- [32] L. Lamport. A Simple Approach to Specifying Concurrent Systems. *Comm. ACM*, 1989.
- [33] L. Lamport. The Temporal Logic of Actions. In *DEC SRC Report 57*, 1990, revised 1991.
- [34] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.
- [35] N.A. Lynch and M.R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of the ACM PODC*, 1987.
- [36] N.A. Lynch and M.R. Tuttle. An introduction to Input/Output automata. Technical report, Technical Memo MIT/LCS/TM-373, November 1988.
- [37] Z. Manna and A. Pnueli. Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs. *Science of Computer Programming*, 1984.
- [38] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.
- [39] M. Merritt, F. Modugno, and M. Tuttle. Time Constrained Automata. In J.C.M. Baeten and J.F. Groote, editors, *CONCUR 91*, 1991. Amsterdam, LNCS 527.
- [40] S.L. Murphy and A.U. Shankar. Connection Management for The Transport Layer: Service Specification and Protocol Construction. *IEEE Transactions on Communications*, 1991. Earlier version in Proceedings ACM SIGCOMM '87 Workshop, Stowe, Vermont, August, 1987.

- [41] S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs – I. *Acta Informatica*, 1976.
- [42] S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM TOPLAS*, 1982.
- [43] D.L. Parnas. A Technique for Software Module Specification With Examples. *Communications of the ACM*, May 1972.
- [44] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of 18th ACM Symposium on the Foundations of Computer Science*, 1977.
- [45] A. Pnueli. The Temporal Semantics of Concurrent Programs. In *Semantics of Concurrent Computation, LNCS 70*, 1979.
- [46] F.B. Schneider and G.R. Andrews. Concepts for Concurrent Programming. In *Current Trends in Concurrency, LNCS 224*, 19XX.
- [47] A.A. Schoone. Verification of Connection-Management Protocols. Technical report, Rijksuniversiteit Utrecht, August 1987. RUU-CS-87-14.
- [48] A. U. Shankar and S. S. Lam. A Stepwise Refinement Heuristic for Protocol Construction. *ACM TOPLAS*, 1992. Earlier version appeared in *Stepwise Refinement of Distributed Systems, LNCS 430*, Springer-Verlag, 1990.
- [49] A.U. Shankar. Verified Data Transfer Protocols with Variable Flow Control. *ACM Transactions on Computer Systems*, 1989.
- [50] A.U. Shankar and S.S. Lam. An HDLC Protocol Specification and Its Verification using Image Protocols. *ACM Transactions on Computer Systems*, 1983.
- [51] A.U. Shankar and S.S. Lam. Time-dependent Distributed Systems: Proving Safety Liveness and Real-time Properties. *Distributed Computing*, 1987.
- [52] G. Tel. Assertional Verification of a Timer-based Protocol. Technical report, Rijksuniversiteit Utrecht, 1987. RUU-CS-87-15.
- [53] G. Tel, R.B. Tan, and J. van Leeuwen. The Derivation of Graph Marking Algorithms from Distributed Termination Detection Protocols. *Science of Computer Programming*, 1988.
- [54] ZZZ. More recent references. X, XXXX.