

Chapter 6

Services and Compositionality

6.1 Introduction

In layered compositionality, a composite system consists of layers of component systems, with services describing the acceptable sequences of interactions between systems in different layers. Figure 6.1 illustrates two kinds of layered systems. The ‘linear’ kind is the simplest, where the component systems are in a column and each one offers the service above using the service below. The ‘mesh’ kind is more general, where each component system offers zero or more services above using zero or more services from any layer below.

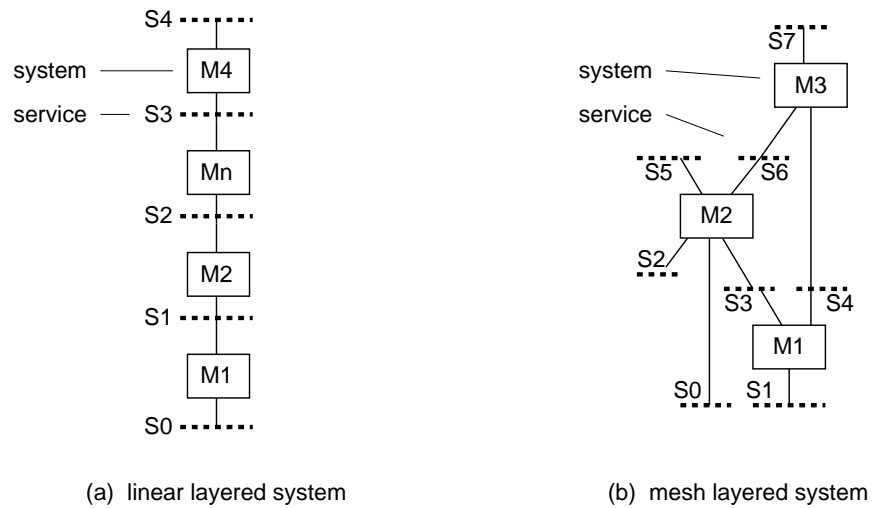


Figure 6.1: Examples of layered composite systems.

Services are expressed by service programs. The purpose of a service program is (1) to define the acceptable sequences of calls of events on one side by the other side, and (2) to be easily understandable and directly usable in analysis and testing. The systems above are referred to as the **users** of the service, and the systems below are referred to as the **offerers** of the service.

Our approach to achieving this is to have the service program be a program with only xc-like events corresponding to the system events of each side that are callable by the other side. Service events are classified into two types: **upward** events, or **upw** events for short, and **downward** events, or **dnw** events for short. Upw events correspond to xc events of systems above the service callable by systems below the service. Dnw events correspond to xc events of systems below the service callable by systems above the service. Service events do not create processes or call events.

The **mapping** of a service event to its corresponding system event can be specified in various ways. Our approach is to rename the system event (via the ‘maps’ attribute) to have the service event name, and

to have other systems call the system event by the service event name. An alternative approach would be to specify the mapping in the service program, for example, by renaming the service event name to be the corresponding system event name. The first approach allows the service to be independent of the system event name and call mechanism (e.g., system call or function call). But it requires that the maps renaming be effected at run-time or earlier.

Whatever approach is chosen to indicate the correspondence, a service event and its corresponding system event have the same signature modulo the name mapping. Furthermore, *the signature is all that the service event and the system event have in common*. The variables, enabling conditions and actions of the service program are solely intended to define the acceptable sequences of interactions; specifically, the event traces of the service program’s executions are themselves the acceptable sequences of event calls. Service programs are not intended to be executed, except in testing, and hence are not bound by the computational and atomicity constraints of the underlying execution platform. It is precisely this freedom from “physical-world” constraints that makes a service program easier to understand than a system program that offers the service.

Below, we describe service programs. We formalize the notion of a system satisfying its services, first in terms of system and service executions, and then in terms of the system and service programs. We establish compositionality, that is, in a layered composite system, if each component system satisfies its services in isolation, then the entire composite system satisfies its services.

6.2 Service programs

A **service program** has the structure

```
service-program <name>( <parameters> ) {
  // no process, blocking construct, or event call
  <initialization>      // constants, types, variables, functions
  <upward events>      // upw events
  <downward events>    // dnw events
  <progress obligations>
}
```

A service event has the structure

```
dnw-event | upw-event <return-type> <event name>( <event parameters> )
ec <enabling condition>
ac <action>           // no event call, no process creation, no blocking
```

The header indicates the event’s signature, consisting of a tag indicating whether it is upward or downward, return type (which may be absent), event name, and parameters (if any) and their types. The event corresponds to an xc event with the same signature (modulo name mapping). For events with return types, the action has return statements. The action does not make event calls, have blocking statements, or create processes. Because the service program is not intended to be implementable, we are not concerned with the computational cost of the action or enabling condition; for example, it may solve an NP-hard problem in variables which, in an implementation, would be spread over different sites.

The progress obligations of the service define the progress that is expected in executing upw events. It has the structure

```
progress-obligations { < leads-to assertions> }
```

The service program defines a set of **executions**, obtained by treating the program as a system program with all events being xc events. The **complete executions** are the executions that satisfy the progress obligations. The event traces of the executions define the acceptable sequences of interactions between systems above and below. The complete event traces, that is, the event traces of the complete executions, are those in which the offerers have fulfilled the progress obligations.

A service program must satisfy certain **consistency conditions**. First, it must be fault-free (otherwise, its executions are not defined). Second, its progress obligations must be **upward-realizable**, which means that every finite execution of the service that does not satisfy the progress obligations can be extended by a sequence of upw events to an execution that does satisfy the progress obligations. Upward-realizability ensures that the service does not expect the users to execute dnw events. To illustrate, $P \rightsquigarrow Q$ is upward-realizable if whenever P holds there is an upw event enabled whose execution establishes Q (does the converse hold?).

Multi-user services: Most services in the concurrent world support multiple users simultaneously. A lock service is one example. There are multiple users, each of which can request, acquire, and release the lock. A network-layer data transfer service (e.g., IP) is another example. There are multiple users, each identified by an address. A user can send messages addressed to remote users and receive messages addressed to itself. Because of features such as anonymous sending and multicasting, we cannot treat this service as a collection of independent point-to-point services. The service provided by a typical file system is yet another example. It allows multiple users to create, modify, delete, and share files. Because users can share files, we cannot treat this service as a collection of independent per-user services.

A multi-user service often, but not necessarily, allows a user to safely use the service based only on its past interactions with the service, that is, without requiring it to first synchronize with other users through some means other than the service. Such a multi-user service is said to be “locally usable”. To formalize this, define the **event trace at user i** of a sequence x to be the sequence of event calls and returns in x associated with user i . A dnw event e of multi-user service S is **locally usable** iff, for any two finite executions x and y of S having the same event trace at e ’s user, e is enabled at the end of x iff e is enabled at the end of y . A multi-user service S is **locally usable** iff each of its dnw events is locally usable. (A sufficient condition for event e to be locally-usable is that e ’s enabling condition consists only of variables that are updated only by dnw and upw events associated with e ’s user.)

Nondeterministic actions Should we allow nondeterministic code in service programs, that is, event actions with the selection operator “[]”? It turns out that general nondeterministic code is not needed in service events. Indeed, it makes service programs harder to understand in general, and complicates the program-level formulation of service satisfaction.

But we cannot entirely avoid nondeterministic code. Consider a service event that corresponds to an xc-with-return event. Clearly, the service event also has to return a value. But there is more to it. The xc-with-return event can return a deterministically selected value, whereas the service event may have to return an arbitrary value from out of several possible ones. For example, if the xc-event returns “yes” or “no” based on information (e.g., resource availability) that is not part of the service, the service event has to return “yes” or “no” nondeterministically. The need for such nondeterminism is obvious when one recalls that an event return is just an event call in reverse, and that practically any concurrent service requires nondeterministic selection of enabled events.

Hence, we restrict nondeterministic code in service events to returns of the form $\text{return}(e_1 \llbracket e_2 \rrbracket \cdots \llbracket e_n \rrbracket)$, or the more general $\text{return}(\llbracket i : e_i \rrbracket)$, where the $\{e_i\}$ are expressions that have no side effect and the statement returns the value of an arbitrarily chosen e_i (of course, the service event action can depend on the returned value, as in if $\text{return}(1 \llbracket 3 \rrbracket) = 1$ then \cdots). We refer to such returns as **nondeterministic returns**. As a consequence, *distinct executions of a service have distinct external traces*. In technical jargon, a service program can have *external* nondeterminism but not *internal* nondeterminism.

6.3 Systems encapsulated by services

By default, a system’s inputs are the signatures of its xc events (after any renaming due to the maps attributes), and its outputs are the signatures of event calls that it makes to xc events in the environment. This can be constrained by services in various ways, as we explain next. For notational convenience, define the **null service** to be the service with no events or variables. This comes in handy for specializing definitions.

Consider a system M and a service S . Define M **maps S above** to mean that no system in the environment of M maps dnw events of S or calls upw events of S . Define M **maps S below** to mean that no system in

the environment of M maps upw events of S or calls dnw events of S . Define M **encloses** S to mean that M has two components such that one of them maps S below and the other maps S above.

For a system M and distinct services U and V , define M is **encapsulated by U above and V below** to mean that (1) every input of M is a U dnw event or a V upw event, and (2) every output of M is a U upw event or a V dnw event. So M interacts with its environment only via U and V . This reduces to “ M encapsulated by U above” if V is the null service, and to “ M encapsulated by V below” if U is the null service.

Only the signatures of the events of U and V are relevant for mapping, enclosing, and encapsulation. So these definitions extend naturally to the case where U and V are arbitrary subsets of upw and dnw events from one or more services. This has several uses, one of which is to limit the extent to which one system can access another system. For example, figure 6.2 shows a distributed system M spread over locations A and B , with an upper service U and a lower service V (for example, A and B are IP addresses and V is the IP service). Let M_A and M_B be the components of M at A and B , respectively. To impose the (often desired) constraint that M_A and M_B communicate only through V , one can require that M_A be encapsulated by the events of U and V at A , and M_B be encapsulated by the events of U and V at B .

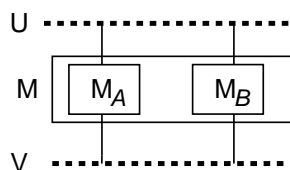


Figure 6.2: A distributed system M encapsulated by services U above and V below.

6.4 Service satisfaction and compositionality

Roughly speaking, a system *satisfies* services above and below if it is encapsulated by the services, makes only safe service event calls, and satisfies the progress obligations of the services above, assuming that the environment makes only safe service event calls and satisfies the progress obligations of the services below. We formalize this next for the linear organization, where a system has at most one service above and one service below, and establish compositionality.

Let S be a service. Let x be a sequence of states, event calls and event returns. We refer to an event call in x as an S -event call if the concerned event is an event of S (modulo name mapping). We refer to an event return in x as an S -event return if it is the return of an S -event call. Define $\text{proj}(x, S)$ to be the sequence of S -event calls and S -event returns in x . In particular, if x is an execution of a system then $\text{proj}(x, S)$ is the sequence of S calls and returns that the system has participated in during x , and if x is an execution of S then $\text{proj}(x, S)$ is the event trace of x .

We say x is **safe with respect to S** , abbreviated x is **safe wrt S** , to mean that there is an execution y of S such that $\text{proj}(x, S)$ is a prefix of $\text{proj}(y, S)$. In particular, if a system that maps S has so far evolved according to execution x , then it can safely participate in the execution of an S event e iff x concatenated with e is safe wrt S .

We say x is **complete with respect to S** , abbreviated x is **complete wrt S** , to mean that there is a complete execution y of S , i.e., an execution satisfying S 's progress obligations, such that $\text{proj}(x, S)$ and $\text{proj}(y, S)$ are equal. In particular, if a system that maps S has so far evolved according to execution x , then it currently satisfies the progress obligations of S .

For notational convenience, we say x is safe (complete) wrt S and T , to mean that x is safe (complete) wrt S and x is safe (complete) wrt T . We say that an execution x of a system M is complete wrt M to mean that it is complete, i.e., satisfies M 's fairness assumptions. This allows us to say, for example, that x is complete wrt M and S .

Definition (service satisfaction (linear)) Given system M and distinct services U and V , we say M **satisfies U above and V below**, also said as M **offers U uses V** , if

- **Signature:** M is encapsulated by U above and V below.

- **Input safety:** For every finite execution x of M such that x is safe wrt U and V , and for every call e of a U dnw event or a V upw event: if $x \circ \langle e \rangle$ is safe wrt U and V , then M has a corresponding xc event that is enabled in the last state of x and whose action executes atomically (i.e., fault-free and terminates). If the xc event's execution returns a value, say g , then $x \circ \langle e, g \rangle$ is safe wrt U and V .
- **Output and internal safety:** For every finite execution x of M such that x is safe wrt U and V , and for every locally controlled event that is enabled at the last state of x : the event execution is fault-free, and if it makes an output call f then $x \circ \langle f \rangle$ is safe wrt U and V .
- **Progress:** For every execution x of M such that x is safe wrt U and V : if x is complete wrt M and V , then x is complete wrt U .

Furthermore, this definition holds even if U and V were to have internal nondeterminism. ■

The second condition implies that M is ready to accept any input that would keep the behavior safe wrt U and V . The third condition implies that any output done by M keeps the behavior safe wrt U and V . As long as the environment does not initiate an unsafe input, every possible evolution of M is fault free and safe wrt U and V . The fourth condition says that every possible evolution of M satisfies U 's progress obligations, provided the environment below satisfies V 's progress obligations and the underlying platform satisfies M 's fairness assumptions.

The above definition handles the case where M satisfies a single service, either U or V , simply by setting the other service to the null service. If V is the null service, we have “ M satisfies U above”, or “ M offers U ”. The above three conditions simplify in that there are no V events (so M is encapsulated by U above) and no V progress obligations. Similarly, if U is the null service, we have “ M satisfies V below”, or “ M uses V ”. The safety conditions simplify in that there are no U events (so M is encapsulated by V below), and the progress condition disappears because there is no U progress obligations.

The above definition of service satisfaction is compositional:

Theorem 6.1 (compositionality (linear)) Let M and N be distinct systems, and U , V and W be distinct services. Let M satisfy U above and V below. Let N satisfy V above and W below. Let MN be the composite system $\{M, N\}$ with xc events corresponding to events of V hidden. Then MN satisfies U above and W below. Furthermore, this holds even if the services were to have internal nondeterminism. ■

Thus in a linear organization of systems and services, if each system satisfies its services above and below then the entire composite system satisfies its services above and below, as illustrated in Figure 6.3. Theorem 6.1 is proved later. Note that the theorem does not say anything about MN with regard to the enclosed service V . In fact, MN satisfies V “internally”, and this will be formalized in the proof.

Strong service satisfaction

The above definition of M satisfies U above and V below requires M to be fault-free only if the environment behaves safely wrt U and V . However, when designing a system M that satisfies U above and V below, we usually prefer that M be fault free in isolation, i.e., as long as the environment calls its xc events only when enabled, regardless of whether that input is safe wrt the service being satisfied. This effects a very desirable separation of concerns: the system integrator can analyze for service satisfaction without worrying about fault-freedom, and the system implementor can analyze for fault-freedom without worrying about the intended services.

Definition (strong service satisfaction (linear)) Given system M and distinct services U and V (which can have internal nondeterminism), M **strongly satisfies U above and V below** if

- **Signature:** M is encapsulated by U above and V below.
- **Fault-freedom:** M is fault-free in isolation (i.e., every finite execution x of M is fault-free).

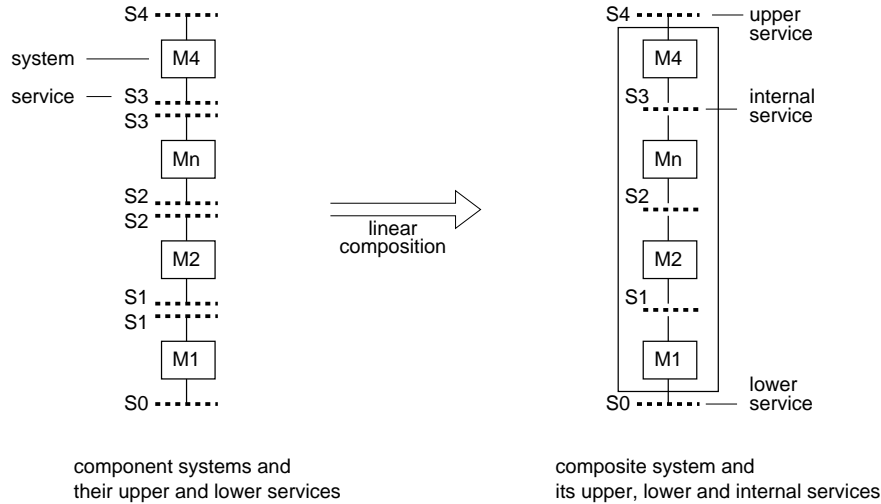


Figure 6.3: A linear layered composite system.

- **Input safety:** For every finite execution x of M such that x is safe wrt $\{U, V\}$, and for call e of a U dnw event or a V upw event: if $x \circ \langle e \rangle$ is safe wrt $\{U, V\}$, then M has a corresponding xc event that is enabled in the last state of x . If the event's execution returns a value, say g , then $x \circ \langle e, g \rangle$ is safe wrt $\{U, V\}$.
- **Output safety:** For every finite execution x of M such that x is safe wrt $\{U, V\}$: if M can make a call f of a U upw event or a V dnw event, then $x \circ \langle f \rangle$ is safe wrt $\{U, V\}$.
- **Progress:** For every execution x of M such that x is safe wrt $\{U, V\}$: if x is complete wrt $\{M, V\}$, then x is complete wrt U .

■

Obviously, strong satisfaction implies satisfaction. But is it still compositional? That is, given distinct systems M and N and distinct services U , V , and W , let M strongly satisfy U above and V below and N strongly satisfy V above and W below. We know that MN , the composite system $\{M, N\}$ with V events hidden, satisfies U above and W below. But does MN strongly satisfy U above and W below?

6.5 Program-based service satisfaction

Consider a system M encapsulated by service U above and service V below. The previous definition of M satisfies U above and V below is stated in terms of the executions of M , U , and V . Although conceptually simple, it does not lend itself to program verification or testing. We now develop an equivalent formulation in terms of the programs of M , U , and V . *Unlike the previous results, the result here requires services without internal nondeterminism* (i.e., events have no nondeterministic code other than nondeterministic returns).

The idea is to define an *artificial* composite system of M , U and V so that every input or output call of M is “coupled” with an execution of the corresponding service event in U or V . Then M , U and V stay “in synch”, and so whether a call is safe is determined exactly by whether the corresponding service event call's enabling condition holds. Below, for any event call $e(x)$ of M , let $e_M(x).ec$ and $e_M(x).ac$ denote, respectively, the enabling condition and action of event $e(x)$ in M . If $e(x)$ corresponds to an event of U , let $e_U(x).ec$ and $e_U(x).ac$ denote, respectively, the enabling condition and action of the U event. Similar notation holds if $e(x)$ corresponds to an event of V .

Consider an M output call $e(x)$ that corresponds to a upw event of U . The call is safe iff $e_U(x).ec$ holds. In reality, the call maps to an xc event of a system in M 's environment. But in this artificial system, let it map to the $e(x)$ event of U , which we now treat as an xc event of U (rather than a upw event). Whenever the call is unsafe, $e_U(x).ec$ would not hold and U would make a faulty transition. This also works if $e(x)$ returns

a value; in this case, $e_U(x).ac$ returns a value which should be accepted without fault by the lc event of M that is called $e(x)$.

Consider an M input call $e(x)$ that corresponds to a dnw event of U . Whenever $e_U(x).ec$ holds, the input call $e(x)$ is safe wrt U , and hence $e_M(x).ec$ should hold and $e_M(x).ac$ should execute atomically (terminate without fault). So if we treat $e_U(x)$ as an lc event that calls $e(x)$ of M (that is, $e_U(x).ac$ becomes $e_U(x).ac; M.e(x)$), then M will make a faulty transition whenever it is not ready to handle a safe input call $e(x)$ (that is, if $e_M(x).ec$ does not hold or $e_M(x).ac$ executes with a fault or does not terminate).

Now consider the case where the input call $e(x)$ returns a value. In this case, $e_M(x).ac$ and $e_U(x)$ each returns a value. We need to check that the two values are equal. This can be done by replacing each $return(z)$ in $e_U(x)$'s action by $if\ M.e(x) \neq z\ then\ fault$. This suffices if z is deterministic. But if z has nondeterministic selection, we need to check that the value returned by $e_M(x)$ is equal to one of the values that z can return. So in the general case, we replace $return(z)$ by $if\ \neg consistent(M.e(x), z)\ then\ fault$, where $consistent(y, z)$ is a boolean function that is true iff y returns a value that equals one of the possible values of z . If z is deterministic, then $consistent(y, z)$ is just $y = z$.

We refer to the modified U as the **system version of U with respect to M** , or $U\text{-wrt-}M$ for short. Similar modifications are done to the service program V , resulting in $V\text{-wrt-}M$. The composite system of M , $U\text{-wrt-}M$, and $V\text{-wrt-}M$ becomes faulty whenever M does not accept a safe input or executes an unsafe output. For M to satisfy U and V , it suffices if the composite system is fault-free and satisfies the progress obligations of U assuming the progress obligations of V .

We now summarize the above (theorem 6.2 below is proved later).

Definition (service-wrt-system) If system M maps service S above then the system $S\text{-wrt-}M$ is defined to be S with the following changes:

- Change “service” to “system” in the program header.
- For every upw event $e(x)$:
 - Change the event type to “xc”.
- For every dnw event $e(x)$:
 - Change the event type to “lc”.
 - If $e(x)$ has no return type, change the action to $e(x).ac; M.e(x)$.
 - If $e(x)$ has a return type, replace each $return(z)$ in the action by $if\ \neg consistent(M.e(x), z)\ then\ fault$.
- Set the fairness assumptions to null.

If M maps S below, then $S\text{-wrt-}M$ is defined the same except that “dnw” and “upw” are interchanged. ■

Theorem 6.2 (program-based service satisfaction (linear)) Let system M be encapsulated by service U above and service V below. Let M^* be the closed composite system $\{M, U\text{-wrt-}M, V\text{-wrt-}M\}$.

- The safety condition for M satisfies U above and V below holds iff M^* is fault-free.
- The progress condition for M satisfies U above and V below holds iff M^* satisfies the progress assertion $V.progress \Rightarrow U.progress$ (or equivalently, every execution of M^* satisfies $(M.fairness \wedge V.progress) \Rightarrow U.progress$).
- The safety condition for M strongly satisfies U above and V below holds iff the following hold:
 - Fault-freedom: M is fault-free (in isolation).
 - Input safety: M^* satisfies
 - * for every U dnw event $e(x)$: $\Box (U.e(x).ec \Rightarrow M.e(x).ec)$
 - * for every V upw event $e(x)$: $\Box (V.e(x).ec \Rightarrow M.e(x).ec)$

- Output safety: M^* satisfies
 - * for every U upw event $e(x)$: $\Box (e_M(x).ec \Rightarrow e_U(x).ec)$
 - * for every V dnw event $e(x)$: $\Box (e_M(x).ec \Rightarrow e_V(x).ec)$

■

6.6 Service satisfaction and compositionality for mesh

The previous section presented service satisfaction and compositionality for linear layered systems. We now extend those results to mesh layered systems, where a component system can have zero or more services above and zero or more services below. The extension is straightforward.

Given a sequence x and a collection of services S , we say x **safe (complete) wrt S** to mean that x is safe (complete) wrt Z for every service Z in S . Given two sets S and T of services, x is safe (complete) wrt $\{S, T\}$ means x is safe (complete) wrt S and x is safe (complete) wrt T .

Definition (service satisfaction (mesh)) Let M be a system. Let U and V be disjoint sets of services. M **satisfies U above and V below**, also said as M **offers U using V** , if

- **Signature:** M is encapsulated by U above and V below.
- **Input safety:** For every finite execution x of M such that x is safe wrt $\{U, V\}$, and for every input call e of M : if $x \circ \langle e \rangle$ is safe wrt $\{U, V\}$, then M has a corresponding xc event that is enabled in the last state of x and whose action executes atomically. If the xc event's execution returns a value, say g , then $x \circ \langle e, g \rangle$ is safe wrt $\{U, V\}$.
- **Output and internal safety:** For every finite execution x of M such that x is safe wrt $\{U, V\}$, and for every locally controlled event that is enabled at the last state of x : the event execution is fault-free, and if it makes an output call f then $x \circ \langle f \rangle$ is safe wrt $\{U, V\}$.
- **Progress:** For every execution x of M such that x is safe wrt $\{U, V\}$: if x is complete wrt $\{M, V\}$ then x is complete wrt U .

■

As usual, this reduces to “ M satisfies U above” if V is the empty set, and to “ M satisfies V below” if U is the empty set. It reduces to the linear case of service satisfaction if U and V each consist of a single service.

Definition (signature compatible (mesh)) Let system M be encapsulated by service sets U_M above and V_M below. Let system N be encapsulated by service sets U_N above and V_N below. M and N are **signature compatible** if

- M and N have no upper service in common, i.e., $U_M \cap U_N$ is empty.
- M and N have no lower service in common, i.e., $V_M \cap V_N$ is empty.
- If M and N have a service in common, then they are in different layers, i.e., at least one of $U_M \cap V_N$ or $V_M \cap U_N$ is empty. (In other words, if an M upper service is an N lower service then no N upper service is an M lower service, and vice versa.)

■

Theorem 6.3 (compositionality (mesh)) Let system M satisfy service sets U_M above and V_M below. Let system N satisfy service sets U_N above and V_N below. Let M and N be signature compatible. Let $S = (U_M \cap V_N) \cup (V_M \cap U_N)$. Let MN be the composite system $\{M, N\}$ with xc events in S hidden. Then MN satisfies $(U_M \cup U_N) - S$ above and $(V_M \cup V_N) - S$ below.

■

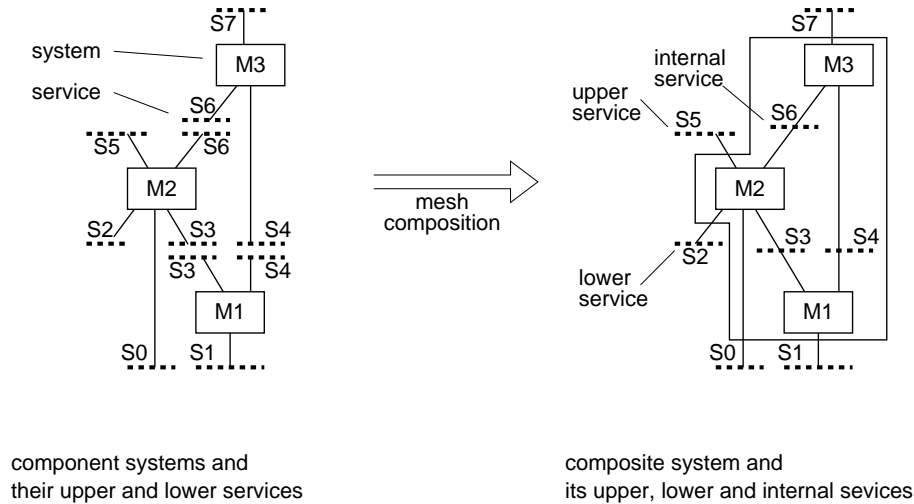


Figure 6.4: A mesh layered composite system.

Thus in a mesh organization of systems and services, if each system satisfies its services above and below, then the entire composite system satisfies its services above and below, as illustrated in Figure 6.4. The proof of theorem 6.3 is the same as that of theorem 6.1, and the details are omitted.

The program-based formulation of service satisfaction is the natural extension of that for linear organization. For a system M and a service set S , let $S\text{-wrt-}M$ denote S with every service Z in it replaced by $Z\text{-wrt-}M$, that is, $S\text{-wrt-}M = \{Z\text{-wrt-}M : Z \text{ in } S\}$. For a service set S , let $S.\text{progress}$ denote the conjunction of the progress obligations of S 's services.

Theorem 6.4 (Program-based mesh satisfaction) Let system M be encapsulated by service sets U above and V below. Let M^* be the closed composite system of M , $U\text{-wrt-}M$, and $V\text{-wrt-}M$.

- The safety condition for M satisfies U above and V below holds iff M^* is fault-free.
- The progress condition for M satisfies U above and V below holds iff M^* satisfies the progress assertion $V.\text{progress} \Rightarrow U.\text{progress}$

■

The proof of theorem 6.4 and its extension to strong-satisfaction is the same as in the linear case. We omit the details here.

6.7 Service satisfaction and compositionality for partial-service mesh

The previous section presented service satisfaction and compositionality for mesh layered systems. We now extend those results to the case where component systems can be encapsulated by partial services.

A partial service arises when a system M uses only a part of a service V , and leaves the rest of V available to be used by other systems. Typically, V is a multi-user service and M corresponds to a subset of users, mapping only those upw events and calling only those dnw events. For example, V may be the lock service and M only accesses the request, acquire, and release events for a particular user.

Although it seems natural for a system to use a part of a service, it does not seem natural for a system to offer a part of a service. The source of this dichotomy is the progress obligations of the service: if a system M accesses only a part of a service U , how does it ensure the progress obligations of U . Indeed, this is not possible if M is a basic system (one without component systems), unless U 's progress obligations are trivial or U consists of several independent services. However, if M has a component M_1 that offers U and a

component M_2 that uses a part of U , then M will, as far as the environment is concerned, offer the part of U not used by M_2 . But there is a crucial difference: the unused part of U is *not* available to the environment of M , unlike in the case where M uses a part of V .

We now formalize the notion of partial services and extend the notion of service satisfaction and compositionality to handle them. For readability, we denote partial services and partial service sets in caligraphic font.

A **partial service** is defined by a service and a nonempty subset of the service's events. For a partial service \mathcal{S} , let $\mathcal{S}.svrc$ denote its service and $\mathcal{S}.eset$ denote its event set. Note that a partial service \mathcal{S} is identical to $\mathcal{S}.svrc$ if $\mathcal{S}.eset$ consists of all events of $\mathcal{S}.svrc$. Thus services are a special case of partial services.

A **partial service set** is a set of partial services such that no two partial services are derived from the same service, i.e., for every two \mathcal{Y} and \mathcal{Z} in the set, $\mathcal{Y}.svrc \neq \mathcal{Z}.svrc$. For a partial service set \mathcal{S} , let $\mathcal{S}.svrc$ denote the set of its underlying services and $\mathcal{S}.eset$ denote the set of its events, i.e., $\mathcal{S}.svrc = \{\mathcal{Y}.svrc : \mathcal{Y} \in \mathcal{S}\}$ and $\mathcal{S}.eset = \cup_{\mathcal{Y} \in \mathcal{S}} \mathcal{Y}.eset$.

We first extend the notions of “safe wrt” and “complete wrt” to partial services. Given a sequence x and a partial service \mathcal{S} , we say x is **safe (complete) wrt \mathcal{S}** to mean that there is an execution (a complete execution) y of $\mathcal{S}.svrc$ such that x and y have the same sequence of $\mathcal{S}.eset$ event calls and returns. Note that y is an execution of $\mathcal{S}.svrc$; \mathcal{S} itself does not have executions. Given a sequence x and a partial service set \mathcal{S} , x **safe (complete) wrt \mathcal{S}** means x safe (complete) wrt \mathcal{Y} for every partial service \mathcal{Y} in \mathcal{S} . Given these definitions, the treatment of partial-service meshes is similar to meshes.

Definition (service satisfaction (partial-service mesh)) Let M be a system. Let \mathcal{U} and \mathcal{V} be distinct partial service sets. M **satisfies \mathcal{U} above and \mathcal{V} below**, also said as M **offers \mathcal{U} using \mathcal{V}** , if

- **Signature:** M is encapsulated by \mathcal{U} above and \mathcal{V} below, and encloses $\mathcal{U}.svrc$ events not in $\mathcal{U}.eset$.
- **Input safety:** For every finite execution x of M such that x is safe wrt $\{\mathcal{U}, \mathcal{V}\}$, and for every input call e of M : if $x \circ \langle e \rangle$ is safe wrt $\{\mathcal{U}, \mathcal{V}\}$, then M has a corresponding xc event that is enabled in the last state of x and whose action executes atomically. If the xc event's execution returns a value, say g , then $x \circ \langle e, g \rangle$ is safe wrt $\{\mathcal{U}, \mathcal{V}\}$.
- **Output and internal safety:** For every finite execution x of M such that x is safe wrt $\{\mathcal{U}, \mathcal{V}\}$, and for every locally controlled event that is enabled at the last state of x : the event execution is fault-free, and if it makes an call f of a $\mathcal{U}.svrc$ or \mathcal{V} event, then $x \circ \langle f \rangle$ is safe wrt $\{\mathcal{U}.svrc, \mathcal{V}\}$.
- **Progress:** For every execution x of M such that x is safe wrt $\{\mathcal{U}, \mathcal{V}\}$: if x is complete wrt $\{M, \mathcal{V}\}$ then x is complete wrt $\mathcal{U}.svrc$.

■

This reduces to the definition of service satisfaction for mesh if every partial service is a full service. Note that an lc event execution of M is expected to be safe wrt $\mathcal{U}.svrc$, rather than just \mathcal{U} , reflecting the fact that M encloses the events of $\mathcal{U}.svrc$ not in $\mathcal{U}.eset$.

Definition (signature compatible (mesh partial service)) Let system M be encapsulated by partial service sets \mathcal{U}_M above and \mathcal{V}_M below. Let system N be encapsulated by partial service sets \mathcal{U}_N above and \mathcal{V}_N below. M and N are **signature compatible** if

- M and N have no upper service in common, i.e., for every \mathcal{Y} in \mathcal{U}_M and \mathcal{Z} in \mathcal{U}_N : $\mathcal{Y}.svrc \neq \mathcal{Z}.svrc$.
- M and N have no overlapping lower partial services, i.e., for every \mathcal{Y} in \mathcal{V}_M and \mathcal{Z} in \mathcal{V}_N : if $\mathcal{Y}.svrc$ equals $\mathcal{Z}.svrc$ then $\mathcal{Y}.set \cap \mathcal{Z}.set$ is empty.
- If M and N have a service in common, then they are in different layers, i.e., at least one of $\mathcal{U}_M.svrc \cap \mathcal{V}_N.svrc$ or $\mathcal{V}_M.svrc \cap \mathcal{U}_N.svrc$ is empty.
- If M has a lower partial service that derives from an upper partial service of N , then M accesses what is available, i.e., for every \mathcal{Y} in \mathcal{U}_N and \mathcal{Z} in \mathcal{V}_M , if $\mathcal{Y}.svrc = \mathcal{Z}.svrc$ then $\mathcal{Y}.eset \supseteq \mathcal{Z}.eset$.

- If N has a lower partial service that derives from an upper partial service of M , then N accesses what is available, i.e., for every \mathcal{Y} in \mathcal{U}_M and \mathcal{Z} in \mathcal{V}_N , if $\mathcal{Y}.svc = \mathcal{Z}.svc$ then $\mathcal{Y}.eset \supseteq \mathcal{Z}.eset$. ■

For any two partial service sets \mathcal{U} and \mathcal{V} , let $\mathcal{U} \ominus \mathcal{V}$ denote the partial service set that remains after removing from \mathcal{U} any events that are in \mathcal{V} . i.e., $\mathcal{U} \ominus \mathcal{V} = \{\mathcal{Y}.eset - \mathcal{Z}.eset : \mathcal{Y} \in \mathcal{U}, \mathcal{Z} \in \mathcal{V}, \mathcal{Y}.svc = \mathcal{Z}.svc\}$.

Theorem 6.5 (compositionality (partial service mesh)) Let M and N be two systems, each encapsulated by a partial service set above and a partial service set below. Let M satisfy \mathcal{U}_M above and \mathcal{V}_M below. Let N satisfy \mathcal{U}_N above and \mathcal{V}_N below. Let M and N be signature compatible. Let MN be the composite system $\{M, N\}$ with xc events in $\mathcal{U}_N.eset \cap \mathcal{V}_M.eset$ and $\mathcal{U}_M.eset \cap \mathcal{V}_N.eset$ hidden. Then MN satisfies $(\mathcal{U}_N \ominus \mathcal{V}_M) \cup (\mathcal{U}_M \ominus \mathcal{V}_N)$ above and $(\mathcal{V}_N \ominus \mathcal{U}_M) \cup (\mathcal{V}_M \ominus \mathcal{U}_N)$ below. ■

Thus, as illustrated in Figure 6.5, in a mesh organization of systems and services, if each system satisfies its services above and below, then the entire composite system satisfies its services above and below. The proof of theorem 6.5 and its extension to strong-satisfaction is similar to the mesh case. We omit the details here.

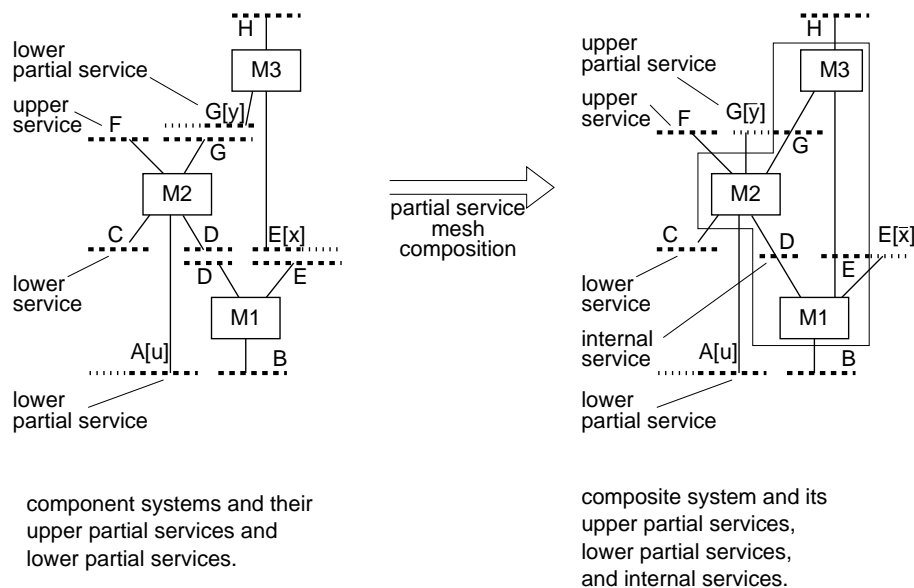


Figure 6.5: A mesh layered composite system with partial services.

Program-based formulation Given a system M encapsulated by a service U above and a partial service \mathcal{V} below, we want to express M satisfies U above and \mathcal{V} below in terms of the programs of M , U , and \mathcal{V} . More precisely, we need to obtain the systems $U\text{-wrt-}M$ and $\mathcal{V}\text{-wrt-}M$. What distinguishes this from the linear case is that M maps only a part of the events of $\mathcal{V}.svc$.

What do we do with an (upward or downward) event $e(x)$ of $\mathcal{V}.svc$ that does not appear in $\mathcal{V}.eset$? We cannot leave out it out, because then $\mathcal{V}\text{-wrt-}M$ would not have all the possible executions of \mathcal{V} and so would not, in general, be able to provide M with all the possible safe inputs. The trick is to make $e(x)$ into an lc event in $\mathcal{V}\text{-wrt-}M$. If $e(x)$ has a return value, we simply ignore it, or, equivalently, remove all return statements in its action.

To summarize, $\mathcal{V}\text{-wrt-}M$ is obtained from \mathcal{V} in exactly the same way as described before, except that every $\mathcal{V}.svc$ event that is not in $\mathcal{V}.eset$ is made into an lc event (so dnw events in $\mathcal{V}.eset$ becomes xc events, and all other events of $\mathcal{V}.svc$ become lc events). If \mathcal{V} is a partial service set, then, as before, $\mathcal{V}\text{-wrt-}M$ is just $\{\mathcal{V}\text{-wrt-}M : \mathcal{Y} \in \mathcal{V}\}$.

Theorem 6.6 Let system M be encapsulated by service set U above and partial service set V below. Let M^* be the closed composite system of M , the systems in U -wrt- M , and the systems in V -wrt- M .

- The safety condition for M offers $U[A]$ using $V[B]$ holds iff M^* is fault-free.
- The progress condition for M offers $U[A]$ using $V[B]$ holds iff M^* satisfies the progress assertion $(M.\text{progress} \wedge V.\text{progress}) \Rightarrow (U.\text{progress})$

■

The proof of theorem 6.6 and its extension to strong-satisfaction is similar to the mesh case. We omit the details.

6.8 Concluding remarks

Service programs define the signatures of the interactions between systems above and below the service, and the acceptable sequences of these interactions. Coming up with a service program is, in general, an incremental activity. One usually starts with some apriori notions and a description of a concurrent system that is supposed to provide the service. One examines the given system and identifies the features that should go into the service; for example, which xc events should be dnw events, which xc events in the environment called by the system should be upw events, which parts of the state to encode in the service, and so on. Not everything in the system should go into the service, otherwise one has a service that is so specific that it rules out other implementations. However, enough has to go into the service so that it is usable by applications, and for this one needs to examine applications that interact with the system.

Services allow one to build systems in isolation such that they will work properly together. If one system is encapsulated above by a service and another system is encapsulated below by the same service, then the two systems are signature compatible. But this only covers individual interactions. For the two systems to work together correctly, we also want the two systems to be compatible in the *sequence* of their interactions over time. This is formalized by the definition of a system satisfying its services. The definition is stated in terms of system and service traces. We also have a program-based formulation which is easier to apply. Service satisfaction is compositional (otherwise it wouldn't be of any use).

We have restricted nondeterministic code in service events to what is essential, namely, nondeterministic returns, thereby ensuring that services have no internal nondeterminism. This restriction does not limit the range of services that can be modeled. It is also not needed for the definition of service satisfaction or the compositionality theorem. But without it, the program-based formulation of service satisfaction would be much more complicated.

6.9 Exercises

- 6.1 Obtain a multi-source single-destination nonblocking channel service program $\text{FifoManyToOne}(N, \text{Msgs})$ as follows. There is a single destination user, associated with id 0. There are N source users, associated with ids $1, \dots, N$. Msg denotes the messages that can be sent. There is a single downward event $\text{Tx}(i, m)$, representing user i sending message m into the service. There is a single upward event $\text{Rx}(m)$, representing the service delivering message m to the destination user. The messages sent by user i , for any i , are delivered in order, but messages sent by different users are delivered in arbitrary order. So if user 1 sends a, b, c and user 2 sends x, y, z , they can be delivered in the order x, a, b, y, c, z .

(Hint: This service is not just N copies of fifo point-to-point channels because $\text{Rx}(m)$ does not have a parameter identifying the sender of message m . It is easy to specify if internal nondeterministic code were allowed.)

- 6.2 Come up with an example demonstrating the need for the nondeterminism constraint in theorem 6.2. Specifically, come up with a system M and a service U with internal nondeterminism such that M offers U but M^* is not fault-free.