Use Case Maps for Object-Oriented Systems

CHAPTER 3

Basic *Use Case Map* Model

This is a self-contained tutorial on the basics of the use case map model that explains the notation and provides rules and guidelines for creating legal maps, interpreting the maps in behaviour terms, binding the maps to components during design, and working with maps at different scales in a coordinated way. It defers issues of concurrent paths in maps to Chapter 7.

The use case map model is a high-level design model to help humans express and reason about a system's large-grained behaviour patterns. The name comes from the fact that maps are a visual notation for use cases [18] and an extension of them into high-level design. However, the model does not depend on knowing what a "use case" is first. The model provides its own definition in its own terms.

There is a trap for the unwary in the parts of the model that relate to *interpreting maps in behaviour terms.* The trap is in looking for more than is intended to be there. There is no intent that use case maps provide complete behaviour specifications of systems. Because maps include some elements that provide cues to behaviour, it is easy to be misled into thinking they are for this purpose. From a behaviour-specification perspective, they are intended only as a framework for humans to use for reasoning and explanation purposes. Maps deliberately leave some decisions open that would have to be made to get a complete behaviour specification. We shall point out along the way specific places where this issue arises, but it is important to keep it in mind from the outset.

3.1 BASIC NOTATION AND INTERPRETATION

The basic symbols of the notation are as follows:

∼ **Path.** A path may have any shape as long as it is continuous. A path may even cross itself, but this can create visual ambiguity related to other aspects of the notation, so the crossing must be distinguished by a small crossover arc or a break in one of the crossed lines.

● **Waiting place**. A filled circle represents a start point in all the examples we have seen so far. In general, a start point is a waiting place for a stimulus to start the path. We use the same symbol for waiting places along paths, for example, to wait for events from other paths.

◷ **Timer**. A timer is a generalized waiting place that expresses the idea that there is a time limit on waiting. It may be used anywhere a waiting place symbol is used.

▌ **Bar**. A bar ends a path or marks a place where concurrent path segments begin or end.

●—▌ **Basic Path**. The most basic, complete unit of a map is a path with a start marked by a waiting place and an end marked by a bar.

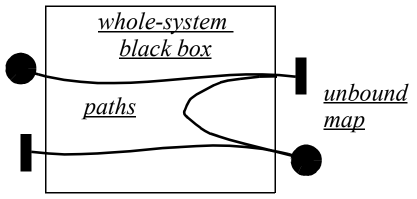
→ **Direction** (optional). Direction is indicated by the positioning of the start and end points but it is sometimes useful to show local direction in a complicated map or in an incomplete fragment of a larger map.

There are a few auxiliary symbols that will be introduced as we go along, but they are not fundamental to understanding the notation.

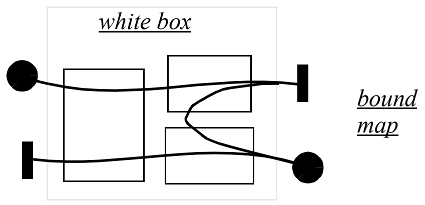
3.1.1 General Nature of Maps

Maps are composed of paths that may traverse components. They are intended to be used at the requirements level and for high-level design.

At the requirements level, a system is composed of one or more large-grained components that are viewed as black boxes (possibly only one, representing the entire system). At this level, the paths of maps traverse the black boxes but do not penetrate them (meaning we do not see internal components or the relationship of the paths to them).The paths are routes along which chains of causes and effects propagate through the system.

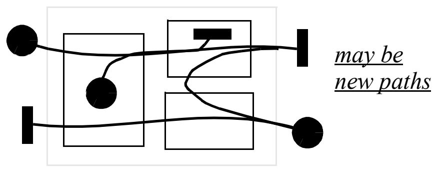


To design a system, we must open up the black box to expose its internal components. A black box with its internal components exposed is often called a white box (the



outline of the white box, shown dotted here, is often left out of such diagrams). One way of characterizing use case maps is to say that they are a means of explicitly linking black-box and white-box views of the large-grained behaviour patterns of systems.

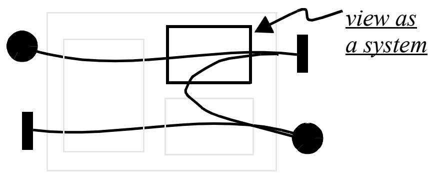
In general, opening up a black box may add new paths that start in internal compo-



nents and either end internally, or follow one of the external path segments to its end. However, this adds nothing new in principle to the meaning of the map model.

Observe that maps with system components visible along their paths are called *bound maps* and ones without (not counting the whole-system box) are called *unbound maps.* If we do not need to make the distinction, the term “map” by itself may refer to either type.

Because components inside a white box may themselves be black boxes, the model may be applied recursively. Later we show how to make a local map for an internal black box from a white-box map that includes it by factoring the white-box map (Section 3.5).



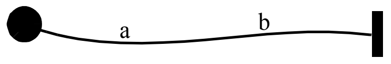
Levels of recursion of black-box and white-box maps are *not* the same as levels of abstraction in design (for example, high-level design, detailed design). Different levels of recursion may be at the same level of design, which would be high-level design as long as we stick with the map model. Detailed design begins when we shift to other models for black boxes at some level of the recursion, like collaboration graphs for interactions between them and state machines for their internal control logic (not covered in this book, but a standard technique).

Unbound maps provide a visual notation for use cases. They are useful for smoothing the transition from black boxes (requirements) to white boxes (design) by enabling a designer to begin with them and add binding and path refinements later as the need for system components is discovered. They can also be useful as reusable patterns that may apply to a range of bindings.

Bound maps are high-level design diagrams that show how a system's components contribute jointly to its large-grained behaviour patterns.

3.1.2 Responsibilities

There may be named responsibility points along any path. Whether or not they are visible



<u>*responsibility points*</u>

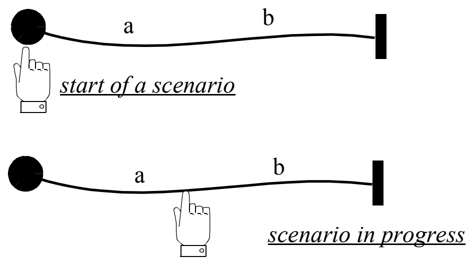
in a particular diagram, the existence of a least one is always implied. A *responsibility* is a named, short, prose description of some localized action a system must perform. By *localized* we mean that responsibilities are viewed as dimensionless *points* along paths. To keep maps as simple as possible, there is no extra notation to mark the points on the path itself; they are indicated only by responsibility names next to the path.

The path chains responsibility points together in cause-effect sequence in relation to the stimulus and a set of preconditions. The original cause is the stimulus. The immediate effect is that the first responsibility along the path is performed. This in turn is a cause relative to the next point along the path after that, and so on as the *causes accumulate to result in each next effect*. The path ends where the ultimate effect is felt. The path is progressive in the sense that each point along it advances the path toward the end. The path may be viewed as representing a *transaction* that must be performed to completion by the system as a whole. Note that just because there is a cause-effect relationship between two responsibilities along *one* path does not mean that there will be a similar relationship between the same two responsibilities along other paths. The cause-effect relationship is a property of each path and the preconditions that cause it, not of the responsibilities.

We say responsibilities are prose descriptions because that is what they are in this book. In principle, there is nothing to stop the notation from being formalized by requiring responsibilities to be expressed in some formal language that links them to changes in the state of the underlying system and, ultimately, to the transformation of preconditions into post conditions by chains of responsibilities. However, the notation is deliberately not formalized in this way in this book.

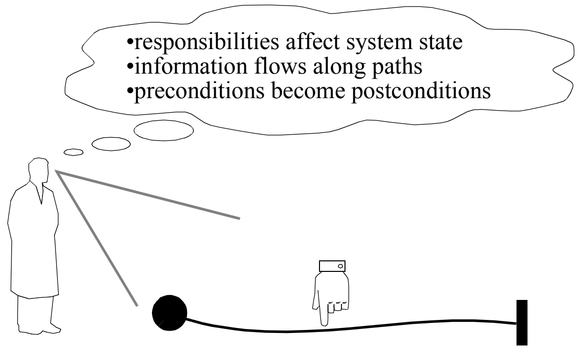
3.1.3 Scenarios and Use Cases

Maps are interpreted in behavioural terms as showing paths for scenarios. Imagine putting a map on a desktop and placing a pointer (symbolized here by —*not* part of the map notation) at the start of a path and then moving it along the path from point to point until the end is reached and the pointer is removed. The path traced is a scenario. We call it a



scenario whether we think of it as being traced by a person as above, or being made to happen by the collaborative action of the components of the system when it is running. In the former case it is a requirement stated by the map. In the latter case it is the observed achievement of the requirement.

The designer thinks about a scenario as showing operation of the system, but the map does not model the way responsibilities change the system state, cause information to flow, and ultimately convert preconditions into postconditions. Responsibilities are just named points on paths.

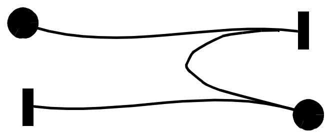

•responsibilities affect system state
•information flows along paths
•preconditions become postconditions

This interpretation of a path as visual representation of a scenario is the link to use cases. A use case is a prose description of a scenario or related set of them, with associated preconditions and postconditions, of a user's interactions with a system seen as a black box. An unbound map is a visual notation for a set of use cases.

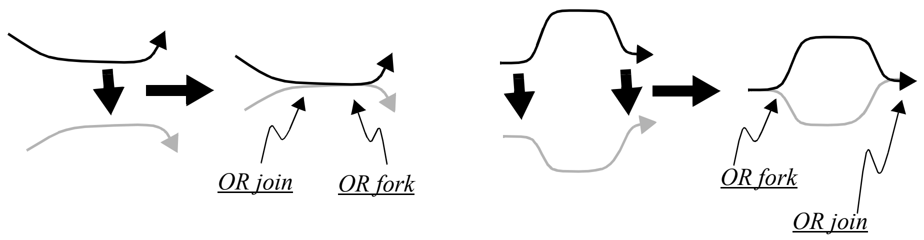
To simplify descriptions, we often use *path* to mean *scenario*, relying on context to make the precise meaning clear. Otherwise, the effort to be more precise leads to wordiness. For example, we speak of a *path* being in progress when we mean a *scenario is in* *progress along a path*, or of *starting a path*, when we mean *starting a scenario down a path*.

3.1.4 Compound Maps

Maps like the ones we used to introduce this chapter are compound, that is they consist of

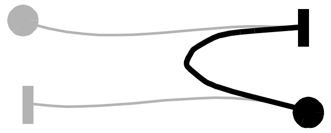


a number of different paths that may or may not be superimposed. The design model view of this is that the end-to-end paths are still distinct and that the points where paths seem to fork and join (called OR forks and joins) are *not* switching points, but only places where the individual paths enter or leave superimposed segments in a diagram. Not being switch-



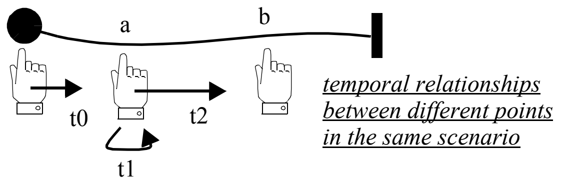
ing points, OR joins are unsynchronized and OR forks need no logic to determine which path to take; the path is implicit in the map to begin with. To identify the different end-to-end paths in diagrams in cases where it is not clear, we use shading , as in this book, or different colours, if drawing on, say, a whiteboard. Otherwise some other scheme must be employed, such as naming path segments and characterizing end-to-end paths by lists of segment names.

To single out for discussion individual paths identifying scenarios we highlight them with contrasting heavier lines.



3.1.5 Real Time Along Paths

Real time (meaning wall clock time here) is implicitly assumed to be required to move

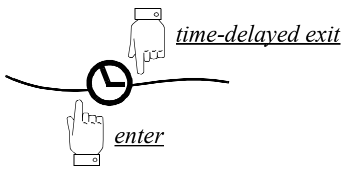

a
b
t0
t1
t2
temporal relationships between different points in the same scenario

along the segments of the paths between responsibility points (for example, t0, t2), and also to progress through each responsibility point (for example, t1). The progression along the path takes time because of communication delays and possibly also because of processing delays in components of an underlying layer that enables progress to take place along the paths. The responsibilities take processing resources and therefore time.

However, this does not necessarily imply that the times are additive, because responsibilities could overlap in time, as follows: Because the intent of use case maps is that responsibilities are large-grained relative to the details of eventual implementations, the possibility exists that responsibilities will be implemented by many fine-grained, concrete actions. Because the intent is that causal linkages along the path segments joining responsibilities are similarly large-grained, the possibility exists that causal linkages may be implemented by many fine-grained, concrete interactions between different components performing different responsibilities. The possibility of many fine-grained, concrete actions and interactions suggests the further possibility that they may be interleaved over time, causing the responsibilities at the use case map level to overlap in time. The only constraints imposed by use case maps are on the relative times when responsibilities may start and end. For example, in the path above, responsibility b cannot start before responsibility a and responsibility a cannot end after responsibility b.

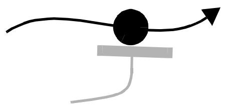
Unbound maps tell us nothing about the sizes of responsibility times or inter-responsibility times. They can only be resolved in bound maps in relation to components, and then only qualitatively as far as the map model itself is concerned. For example, binding responsibility a to a thread in one hardware box and responsibility b to a thread in another would tell us that the ab path is realized by some interprocessor communication hardware and software. This can give us some qualitative feel for the nature of the delay along the path. If we know something about the speeds of the processors and the complexity of the responsibilities in relation to these speeds we would also get some qualitative feel for the total real time along the path in the absence of other activity. The maps themselves are not intended to do more than this for purposes of reasoning about design issues (although in principle they could provide a context for entering performance-related numbers into a performance model).

Real time may also enter maps explicitly. A timer may be used to mark a place where the propagation along a path is delayed by some time period. Because this is not a formally executable model, there is no provision for entering numbers into the map signifying actual timeout periods, and no means of doing anything with the numbers if there were, except displaying them to readers of the map for information. A timer indicates the *existence* of a time delay, not its details. Think of a timer as a special kind of responsibility along a path that takes up real time without taking up processing resources.

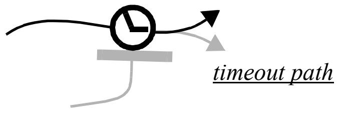


3.1.6 Waiting Places

One of the most important requirements for use case maps is that they provide a means of explicitly showing coupling between scenarios. This is very difficult to do with prose use cases, which tend to bury coupling information in prose descriptions that have to be searched through to answer questions such as: Which other scenarios is this particular one coupled to, and where in the scenarios does the coupling take place? We defer the general issue to Chapter 7, and only explain the simplest form of coupling here. This is the positioning of a waiting place along a path to indicate that it must wait for events along another path. The concept is that the first path pauses until a trigger arrives along the second path.

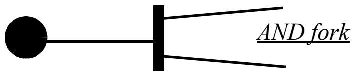


A waiting place may be timed, in which case there may be a need to indicate an alternate timeout path.

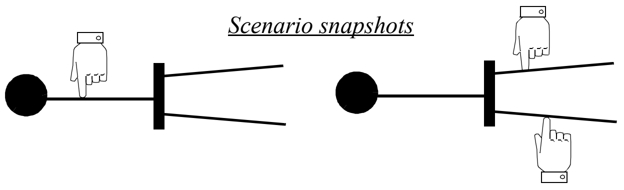


3.1.7 Ripple

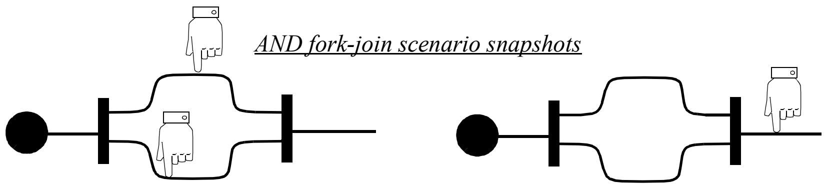
Explicit concurrency in the *same* scenario may be used to express patterns that are not pure point to point, but contain some ripple (recall Chapter 2). This is done by using AND forks. An AND fork is indicated by a bar that splits one entering path into several concurrent forks.



The effect is to split an entering scenario into multiple concurrent parts. The multiple parts progress concurrently in an unsynchronized fashion after the bar, but are still regarded as part of the same scenario. This concurrency may be interpreted to mean don't-care ordering in cases where concurrency is not possible or desired. When we speak of a *path in progress* relative to such cases, we mean the whole path, including the different parts after the AND forks.



A following AND join may be used to end the concurrency. Just as above, the details of what happens when multiple scenarios may be in progress are deferred to components.



3.2 AN EXAMPLE: MOUSE DOUBLE CLICK

At this point it may be helpful to consider a simple example to consolidate these ideas.

Interesting patterns that may be described in a helpful way by use case maps exist at all scales in systems. When we say use case maps are for expressing large-grained patterns in systems, we mean that they are large grained on the scale of whatever system we use them for. However, we do not say what that scale is. Given the recursive black-box/white-

4.6.1 Six Component Types

Figure 4.4 provides a suite of six basic component types for use case maps (a seventh type, dynamic components, never appears directly in maps). This suite is sufficient for the purposes of this book. This figure is not intended to imply that all of these component types must be part of the map model, or that others are excluded. However, we have found these types to be useful across a wide range of system types, programming technologies and applications. *Teams*, *slots*, and *pools* are the *only* uniquely-high-level abstractions we shall need. Teams are used to represent large-grained components (in the sense of operational groupings) or components of uncommitted types. Pools are places to hold dynamic components in readiness to move into slots. Slots are places where dynamic components perform responsibilities. Teams may be fixed components or slots. Dynamic components never appear explicitly in maps because pools and slots are sufficient to express structural dynamics. We add *threads*, *objects* and *interrupt service routines* to round out the model, following the argument in Chapter 1 that we need representations for them that can be used to transition between high-level design and detailed design; these components can also be represented as fixed components or slots. All of these components except pools may be designed to have application-dependent responsibilities. Detailed characterizations of the components of the suite are given following Figure 4.4.

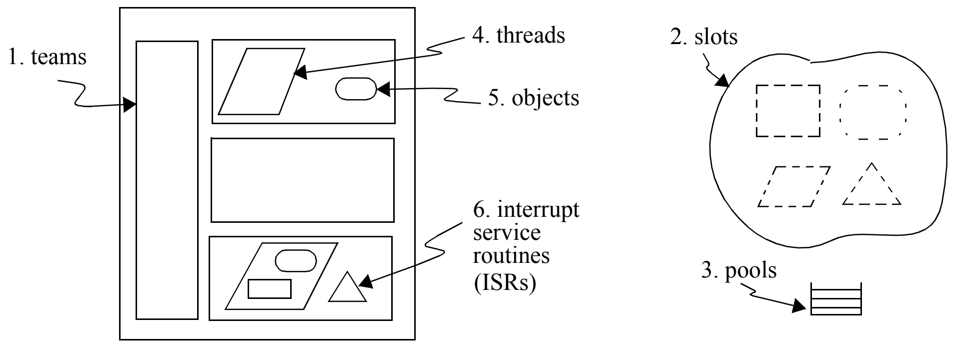
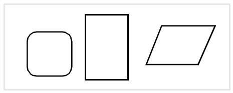


Figure 4.4 Six basic component types for use case maps.

Teams Teams are lightweight abstractions for large-grained components at the level of use case maps. We can use them somewhat casually because no strong commitments are implied by this use. We may introduce them into maps to hide details without committing to whether or not they will actually exist as components with interfaces or will actually have members. Up till now we have used the team notation only to identify the *existence* of components that are operational black boxes, without making commitments to their nature, and we shall continue to use them this way. However, in general, a team is an operational grouping of components that, when opened up as white box, may include as members any or all of objects, threads, interrupt service routines or other teams. A team is said to be *passive* if it contains no threads or interrupt service routines (at any level of recursive decomposition of its members) and to be *active* otherwise.



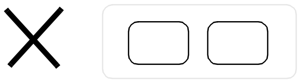
When we speak of a team as a white box or a black box, or of recursive decomposition of a team, all we intend to imply is that its components and their components (and so on, recursively) are or are not visible in certain diagrams. Showing teams in use case maps implies nothing about implementing them as code-level containers. How they are to be implemented is a deferred detail.

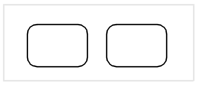
In cases where a team symbol is used only to identify the *existence* of a component without committing to its *type*, the box does *not* necessarily have to be expanded later into more components. We may later reinterpret the team as a component of a specific type either by redrawing the map with the rectangle replaced by another shape, or by putting only one component of the required type in the white box view.

Objects From the behavioural perspective of use case maps, an object is a component that supports a data or procedural abstraction through an interface. Although we *define* an object by saying it supports an abstraction through an interface, the *actual* interfaces of objects are below the level of abstraction of use case maps. The assumption is that the interface—when it is designed—will provide the means for other components to ask it to perform its responsibilities.

The model of behaviour is that objects perform their own responsibilities, but do not have ultimate control of *when* they perform them; the interfaces provide the means for other components to exercise control over "when". Ultimately, this control comes from threads, although it may come indirectly through other nonthread components such as teams or other objects.

Objects are viewed in our design model as fine-grained components that do not have a team property. In other words they are not further decomposable into teams of still finer grained objects. If an abstraction is needed for a coarse-grained "object" that is decomposable into finer grained objects, we use a team.





In our design model there is no raw data, only objects that support data abstractions.

Whether or not a particular object is an instance of a class in an object-oriented class hierarchy is a separate issue.

Threads ▱ Threads are treated in depth in Chapter 7 in association with concurrent maps. Until then, the following will provide a sufficient characterization of them: A thread is an autonomous, self-directed component that may operate concurrently with other threads. Its internal logic is sequential; in other words, there are no concurrent elements inside threads, the only concurrent elements are the threads themselves; multiply-concurrent components are modeled as active teams (teams with multiple threads in them).

Interrupt Service Routines (ISRs) △ Interrupt service routines (ISRs) provide the glue between physical stimuli and other components of the design model, particularly threads. They may often be omitted from high-level designs as lower-level detail. See Chapter 7 for more information.

4.6.2 Relationship to Other Models

All of these component types may appear with the same notation, but with details added, in detailed diagrams of the operation and assembly domains, such as collaboration graphs, visibility graphs and associated interaction sequence diagrams.

Collaboration Graphs and Visibility Graphs Teams are lightweight components at the level of use case maps, but not so lightweight in collaboration graphs and visibility graphs. Commitments must be made about teams in these diagrams, as illustrated by Figure 4.5 and Figure 4.6. Figure 4.5 shows three ways (among others) of representing teams in visibility graphs, all of which imply different implementations: no implementation, a container, or a switchboard. Read the arrows from tail to head as *sees* or *has visibility of*. There is no commitment to actual interfaces, so this is a more lightweight way of representing intended implementation structure than collaboration graphs.

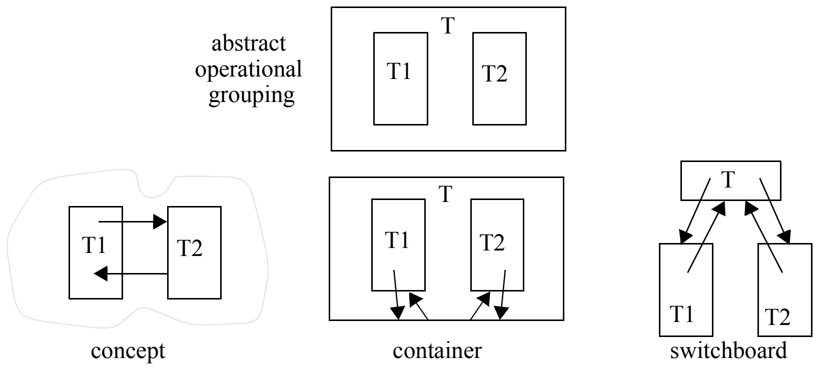


Figure 4.5 Teams expressed in visibility graphs.

C H A P T E R 9

Detailed Design Notation

This chapter provides a general collaboration graph notation that is both particularly simple and particularly widely applicable to a range of object-oriented and real time implementation techniques. It is positioned here to set the stage for detailed developments in the following two chapters. However, except for an overview section, the focus of the chapter is rather detailed and most of it is not needed to understand the essence of the following chapters.

An overview of the collaboration graph notation is provided in Section 9.1; this is new material that gives the essence of the notation. The rest of the chapter is new material for detailed design specialists that may be skipped without loss of continuity in relation to the main ideas of the book.

The collaboration graph notation has specific features for threads, interrupt service routines, objects, slots, and teams. It is also quite simple, employing only five additional symbols

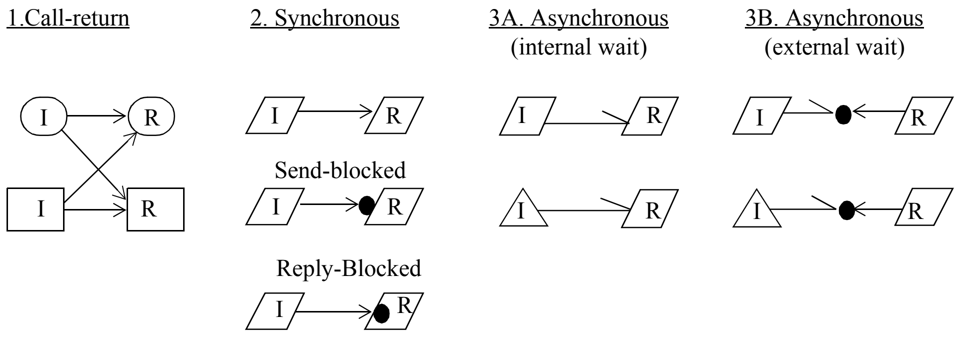
$\longrightarrow$ $\rightharpoondown$ ● ◷ ○→

beyond the components themselves (which are the same ones that have already made their appearance in relation to use case maps). The combination of wide range and simplicity makes the notation interesting and useful in its own right, which is why it is in the body of the book instead of in an appendix (other standard notations used in the book are summarized in Appendix A).

Where use case maps are a high-level expression of the purposeful behaviour of a system of collaborating components, collaboration graphs are a detailed expression of the *interfaces* and *connections* that enable them to collaborate. Our use of the terms “collaboration” and “interaction” is not quite the same. When we speak of a set of components as a whole, we use the term “collaboration”. However, when we speak of activity over a specific connection, we use the term “interaction”.

9.1 THE COLLABORATION GRAPH NOTATION

Figure 9.1 gives the essence of the notation.


1.Call-return
2. Synchronous
3A. Asynchronous (internal wait)
3B. Asynchronous (external wait)
I
R
Send-blocked
Reply-Blocked

Nuances:
- Any component may be a slot (shown by a dashed outline).
- Waits may be plain or timed. ● ◷
- Data flows may be positioned next to compatible connections. ○➝

Legend:
I - Initiator
R - Responder

Figure 9.1 Essence of the notation.

Components at the ends of connections are referred to as *initiators* and *responders*. An initiator is the component that starts any interaction over a connection (for example, makes a call, sends a message). A responder is the component at the other end of the connection from the initiator. The direction from initiator to responder is referred to as the *control* direction. When a connection is composed of a single arrow, the initiator is at the tail and the responder is at the head. However, when the connection is composed of two arrows (asynchronous with external wait), the seemingly counter-intuitive situation occurs that both initiator and responder are at the tails of arrows. This is because both have to go outside themselves to interact over a connection that implicitly has interaction machinery embedded in it that is separate from both the initiator and the responder. This separate machinery is explicitly not modeled by separate components in this notation.

Legal initiators and responders are the same components as before, namely objects, teams, threads, slots, or interrupt service routines. Excluded from the class of initiators and responders are lower-level units of software like functions, methods, semaphores, mailboxes, or entries that in some cases provide the interfaces of components and in all cases are the means of implementing the connections between them. Such units are not separately represented in the notation. They are implied by the connections. Therefore, although simple *arrows* may be easily associated with calls and message sends in sequential software, *connections* may require a deeper interpretation. For example, autonomously generated accepts and replies are implied by synchronous connections, and interaction machinery positioned *between* components is implied by asynchronous ones with external waiting (for example, semaphores, mailboxes).

All we need for design of sequential object-oriented programs are call-return connections. The other connection types are needed for designing real time programs in terms of threads and interrupt service routines.

9.1.1 Attributes of Connections

We will now characterize the connections in terms of the attributes in Figure 9.2, to enable their semantics to be defined in a general way, without reference to specific software implementation technologies. Attributes are assumed to be supplied by the implied interfaces and interaction machinery.

Connection type	Attributes		
	Control	Interaction	Synchronization
Call-return	Initiator	Round trip	Unsynchronized
Asynchronous	Shared	One way	Unilateral
Synchronous	Shared	Round trip	Bilateral

Figure 9.2 Attributes of connections.

Control The control attribute indicates whether or not the initiator and responder ends share control or not. If control is not shared then, by default, it is solely in the hands of the initiator. This means there is nothing to stop multiple concurrent action at the responder end due to multiple concurrent initiators.

Interaction Whether or not control is shared, interactions may be round trip (meaning a response is possible from the responder in the same interaction) or one way (meaning no response is possible in the same interaction).

Synchronization Synchronization types are as follows:

Unsynchronized means there is no synchronization of any kind, because the responder is not capable of participating in it (for example, passive teams, objects).
Asynchronous means unilaterally synchronized, because only the responder may have to wait for an initiator, but never vice-versa.
Synchronous means bilaterally synchronized (both ends have to wait for each other); this is often called *rendezvous*.

9.1.2 Exclusions from the Collaboration Graph Notation

There is no representation for dynamic components except indirectly through slots. This is not an oversight; we rely on other diagrams to convey structural dynamics (use case maps, visibility graphs).

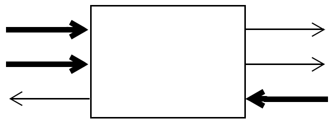
We provide no representation for broadcasting calls or messages to many places at once (analogous to AND forks in use case maps that indicate paths may go in many directions at once). Object-oriented and real time implementation technologies that we aim to cover with this notation do not support broadcasting directly, so we have not felt the need to include it. However, it may be useful to have a notation for broadcasting in relation to distributed operating systems that support it. In such cases, we encourage readers to invent their own.

9.2 USING THE NOTATION FOR DESIGN

9.2.1 General Properties

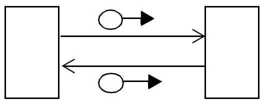
We illustrate the general properties of the notation with team boxes in the following diagrams to signify "any component".

Keep in mind that, although the *system* interface of a component in a collaboration graph is the set of all connections pointing to and away from it (the set of heavy and light arrows in the figure below), the *programming* interface is associated with the set of all



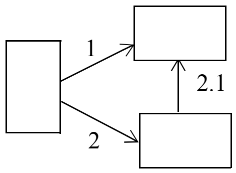
connections pointing *to* it (the heavy arrows only). There may be more to it than this if the component is a slot, because the actual programming interface of a slot occupant may require additional elements to support run-time structural dynamics expressed in maps at a higher level of abstraction.

Data flows may be associated with any connection consistent with the connection



type. They imply parameters of interfaces and interactions. Because data flow is secondary information in collaboration graphs, we often leave it out when giving examples to explain the notation, but it should never be left out of actual design diagrams used for documentation. In general, flows need to be both named and typed. Data flows may be used to provide information about structural dynamics (Section 9.2.2).

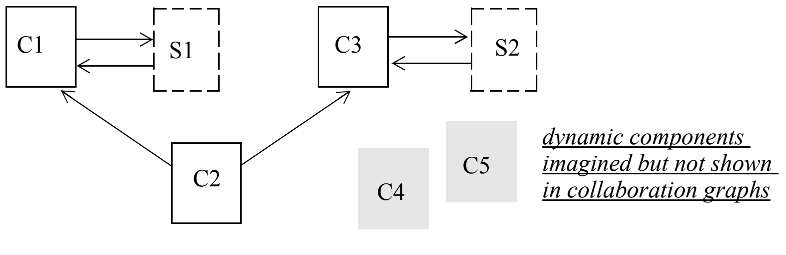
Connections may be annotated with sequence numbers to indicate standard interac-



tion sequences. There is no notation to distinguish connection *names* (which may also be numbers) from sequence numbers, so it must be resolved by context. An indented numbering notation may be used to indicate causally related sequences of interactions in cases where they can be conveyed clearly this way. Otherwise, interaction sequence diagrams may be required to show such sequences.

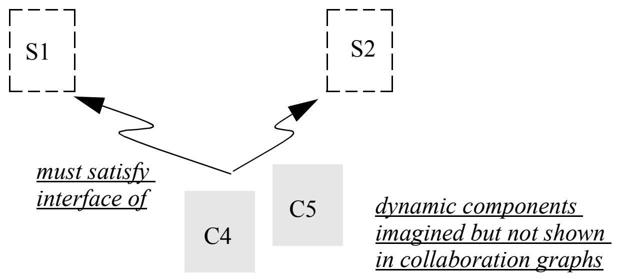
9.2.2 Structural Dynamics

Section 4.6 explained the relationships between slots, dynamic components and classes in terms of visibility graphs and class relationship diagrams. Collaboration graphs do not show dynamic components, but they show slots. Slots are used in collaboration graphs to show different contexts in which dynamic components may be visible at different times, without showing the dynamic components themselves. For example, in the collaboration graph below, the effect of the slots could be to make components like C4 and C5 visible to C1 and C3 at different times. C4 may be in S1 at one time and in S2 at another, and C5 may be in S2 while C4 is in S1 and in S1 while C4 is in S2.

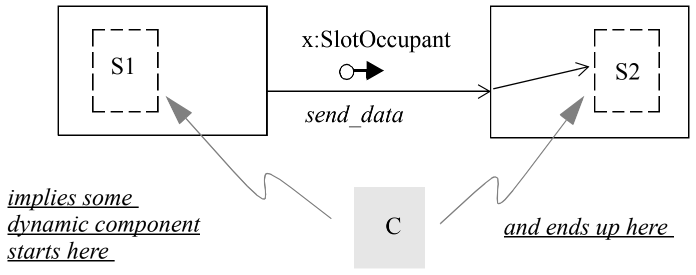

C1
S1
C3
S2
C2
C5
C4
dynamic components imagined but not shown in collaboration graphs

When developing collaboration graphs, we assume that the slots are occupied and ready to interact. Therefore, collaboration graphs normally have no connections in them specifically for creating/deleting components like C4 or C5, or filling slots with them. This is one of the reasons why we said above that the programming interface is only partially determined by collaboration graphs. Leaving out these things actually makes collaboration graphs more useful, because the remaining collaboration patterns are easier to see. Recall that we leave them out not because we think they are secondary scaffolding but because our whole approach to design relies on using maps with slots in them to give a view of structural dynamics at a higher level of abstraction.

The required interfaces of dynamic components come from design diagrams as follows: All dynamic components filling a slot must support the slot interface. Another way of saying this is they must all be of the same type (in a programming-language sense). Dynamic components filling multiple slots must satisfy multiple interfaces. Additional interface elements beyond these may also be required if the dynamic component must be acted on while in transit between slots, to satisfy some responsibility along a use case path (in such case there is no slot to indicate interface elements, they must be derived from responsibilities).


S1
S2
must satisfy interface of
C4
C5
dynamic components imagined but not shown in collaboration graphs

The only place we show dynamic components explicitly in collaboration graphs is adjacent to connection arrows to indicate they are passed as parameters, for example, when a component contains data that is required elsewhere and the whole component is passed instead of just the data.



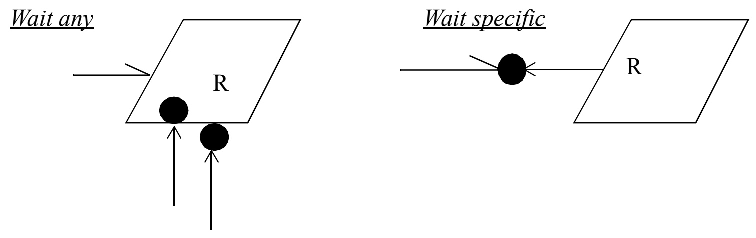
The grand strategy for creating dynamic components and making them visible in different places is best conveyed by a use case map, not by a collaboration graph. This map, together with the collaboration graph, possibly supplemented by visibility graphs, may be used to determine programming details. A considerable amount of detailed programming may be required to manipulate and pass around pointers to ensure that this knowledge is in the right place at the right time. This detail is, in general, below the level of the collaboration graph.

9.2.3 Threads and ISRs

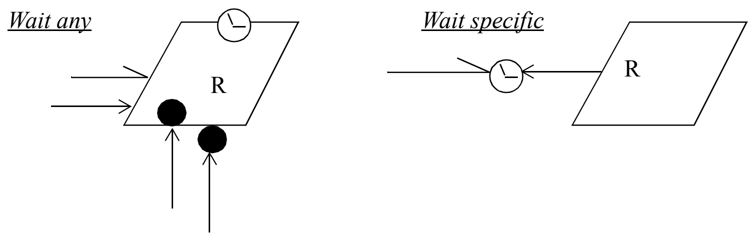
The notation has been designed to express in a generic fashion the essence of a wide range of interthread and ISR-thread interaction mechanisms that exist in practice. Key issues are synchronicity, waiting, and timeout. To model these things we use two styles of arrows and borrow the waiting place and timer symbols from use case maps. The borrowed symbols indicate a more detailed kind of waiting/timeout than for maps, namely waiting for control interactions at interfaces, so there are detailed nuances that do not exist with maps. Otherwise the meanings are consistent. Using the same notation has a beneficial effect once you get used to it because the symbols give essentially the same cues, with nuances obvious from context.

Waiting for Arrivals An *arrival* is something that comes to a responder R at an unpredictable time, like a call, a message or a signal. It may be information it needs or a request for some service. Initiators *cause* arrivals, but are not *themselves* the arrivals.

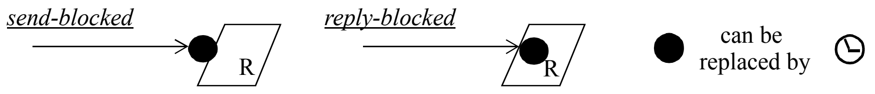
The *waitany* case is the default for R waiting internally for any arrival. A responder may also wait *externally* for an arrival over a *specific* asynchronous connection (remember that the connection in this case is the composition of arrows plus waiting place, and that the individual arrows are not themselves “connections”).



Timed waiting means that R will get either an ordinary arrival before the timeout occurs (in which case the timer is cancelled) or notification of timeout, whichever comes first.



Waiting for Round-Trip Responses (Send-Blocked and Reply-Blocked) Asynchronous interactions do not have round-trip responses by definition, so this only applies to synchronous ones. Waiting place markers are used in a supplementary way to characterize synchronous interactions as *send-blocked* or *reply-blocked* (this adds additional meaning to the nature of the synchronization beyond the bilateral synchronization that is already an inherent attribute of such connections). The supplementary notation indicates that a specific connection is used by the initiator to make a request that may not be immediately satisfiable, forcing it to wait, for example, because satisfying the request requires input from another thread. Note that, although the waiting and timeout symbols are placed at the responder end, they apply to the initiator; this is indicated by associating them with *specific* connections, not the responder thread as a whole.



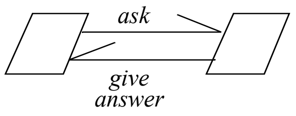
In *send-blocked* interactions, the initiator waits externally, implying that the initiator's request can be ignored by the responder. In this case, the connection itself defines the request. The intent of the timeout is that the initiator may give up after a time, without ever interacting with the responder. Thus the initiator will return from the interaction with either a response or a notification of timeout.

In *reply-blocked* interactions, the initiator waits internally, implying that the initiator's request cannot be ignored, but must be processed by the responder to see what the initiator wants, before determining whether or not an immediate response is possible. The responder may delay a response until it can satisfy the request, perhaps in consequence of a subsequent interaction with some other thread. The meaning of the timeout is that the responder will return an indication of timeout if a timeout period elapses before the response is available. Thus the initiator will return from the interaction with either a response or a notification of timeout.

9.2.4 Connection Patterns

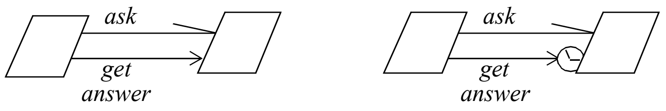
The notation may be used to express connection patterns spanning several connections (patterns of many kinds exist in systems). Sometimes it is useful to have special symbols for these (although we do not show any here).

Asynchronous Request-Response: This is roughly equivalent to a reply-blocked syn-



chronous interaction, except that it allows the thread sending *ask* to do something else while the answer is being prepared. Because there is no round trip in either case, neither knows whether or not the other got anything or acted on it, so this is a bit shaky in an environment where attempts at asynchronous interactions might be lost.

Futures: Here the thread sending *ask* assumes, some time after asking, that the

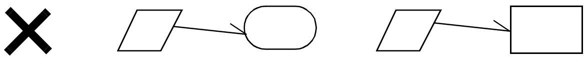


responder will have the answer, and so does a synchronous *get* to get it. The interaction is not shown as reply-blocked because the presumption is that the other thread is ready. Caution might suggest adding a send-blocked style of timeout in case the other thread is, for some reason, unavailable.

9.2.5 Connection Rules

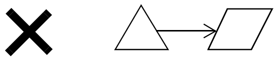
Consistent and Inconsistent Attributes Attributes of connections must be consistent with the components they connect. The following combinations are illegal because of inconsistent attributes:

- This combination is illegal because the connections are asynchronous, but objects

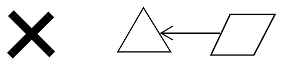
  and teams require connections with an unsynchronized attribute. Replacing the half-sided arrow with a two-sided one renders them legal because the latter is interpreted by context to have an unsynchronized attribute.

- This combination is illegal because the implied synchronization could require the

  interrupt service routine to wait for the thread, which, although it might be programmable, would be dangerous.

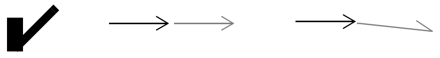
- This combination is illegal because interrupt service routines cannot be called by



software.

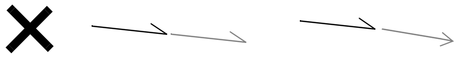
Connection Chains Connections may be chained together between components and across component boundaries. Chains have the combined attributes of their parts; as long as the combination of attributes makes sense, the chain is legal.

- The following chains of arrows are legal; the first because it represents either



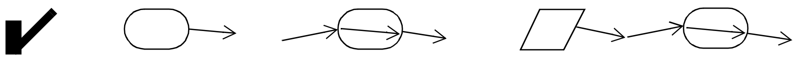
chained call-returns, or a call-return with a synchronous interaction chained off the end; and the second because it represents a call-return with an asynchronous interaction chained off the end. These combinations make sense.

- The following combinations are illegal; the first because asynchronous is asynchronous, no matter how many arrows there are; and the second because there is no path for a round trip if the second arrow is interpreted as call-return (the other interpretation, that this implies an interaction with both asynchronous and synchronous attributes, is meaningless).



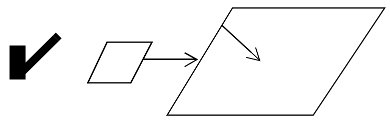
Chaining Connections Across Interfaces Connections may be chained across component boundaries to show external-internal connections, where appropriate. The attributes of the connections in relation to the components determine legal and illegal combinations:

- Any outgoing arrow from an object implies an outside initiator, chained through the



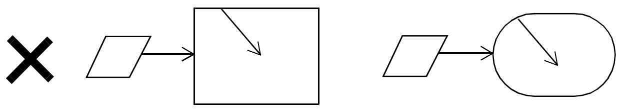
object. The ultimate initiator must be a thread (or ISR), somewhere at the start of a continuous chain leading to the object, whether or not it is shown in the particular figure. This thread (or ISR) is required to act as the ultimate initiator, because the object itself is not an autonomous component. Concurrent programs may have many ultimate initiators, sequential programs may have only one (for a sequential program, the single initiator thread would correspond to the mainline).

- Discontinuities in the chain between internal and external connections of a thread



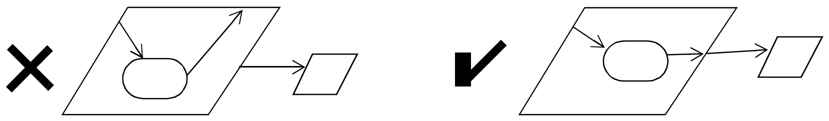
indicates the thread acts as an autonomous initiator for the internal connections at times of its own choosing.

- Discontinuities across team or object boundaries are meaningless because neither

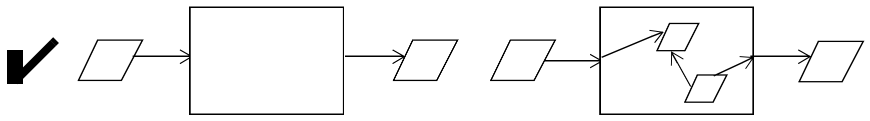


can be autonomous initiators in their own right.

- Discontinuities across thread boundaries of connection chains originating in objects that are operationally internal to the threads are illegal because they would imply that the thread is interacting with itself (implicitly, the interaction would be chained through the internal object in the figure).



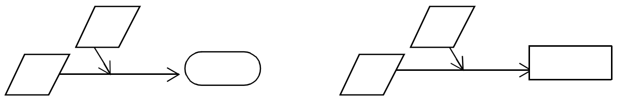
- If a team contains threads, the connection chains crossing the boundary may have



different attributes as a whole from the parts that appear outside. This may give a misleading impression of connection attributes if the team is shown as a black box. It is not wrong to show such a team as a black box, but diagrams that aim to give an accurate picture of the connection attributes of the whole need to expose internal threads.

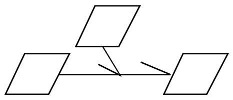
Fan In and Fan Out The fact that connections implicitly contain collaboration machinery has implications for fan in, as described below. The fanned-in connections are shown offset in the diagrams below, for visual clarity, but the meaning is the same as if all connections were joined at the same point on the responder. The interesting fan in issue arises when the initiators are threads, either directly, or indirectly through some cascaded set of call-return connections. (If you use thread stacks, remember that connections that are drawn only to the top actually represent fan in to, or out from, all.)

- Fan in of *call-return* connections from different threads implies possibly concurrent



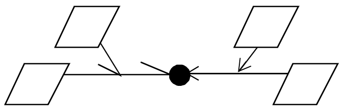
  execution at the responder. There is no implied mutual exclusion, serialization, or queuing.

- Fan in of *asynchronous* connections from different threads implies nothing new.



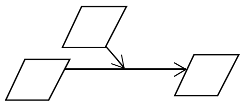
  Arrivals are implicitly queued in first-in-first-out order at the responder, whether there is fan in or not. The responder implicitly dequeues them one at a time, waiting if nothing is there.

- Fan in from different threads may occur at the both ends of asynchronous connec-



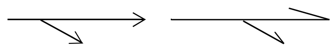
  tions with external waiting. There is nothing new at the initiator end, but, at the responder end, it implies multiple responders may have to wait in first-in-first-out order to get arrivals.

- Fan in of *synchronous* connections from different threads implies that *both* arrivals



  and initiators are queued in first-in-first-out order. Arrivals are assumed to be dequeued in this order. Initiators are dequeued in this order to return with responses. There is assumed to be no possibility of responding in different order to initiators for the *same* connection (although there is *between* connections).

- Fan out always means that the initiators interact with the connections one at a time



in some sequential order determined by their internal logic.

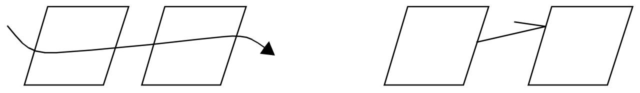
9.3 COLLABORATION GRAPHS AND USE CASE MAPS

Here we provide some guidelines for inferring relationships between local patterns in maps and ones in collaboration graphs. Don't be misled by the fact that *local map patterns* and *local communication patterns* appear to be at approximately a similar level of visual complexity. The appearance is misleading because end-to-end information that is in the map as a whole is lost in the collaboration graph. This is not important for fragments like these, but it is important for whole systems. When going from maps to collaboration graphs during design, the lost information has to be added back via the internal logic of the components. But this loses the big-picture view that is explicit in maps. To recover the big picture requires putting together a lot of scattered detail. The notations have different purposes and in fact are complementary at different levels.

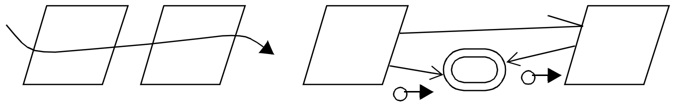
9.3.1 One-Way Paths and Excursions

This section illustrates how one may use communication mechanisms to control overtaking and congestion along paths. Generally, there are many possibilities along one path, and more than one has to be considered to make an informed design decision.

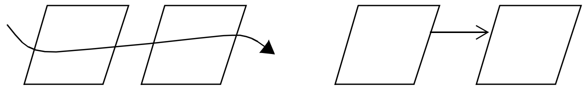
One Way Paths The minimum connection required for a *one-way* path is a single asynchronous one in the direction of the path. Such a connection enables maximum concurrency along the path, because the first thread does not have to wait for the second one. In other words the rate of progress along the path is determined by the first thread. However, it is also dangerous because the second thread could be overwhelmed. The communication mechanism is assumed to provide queuing but the queue could overflow.



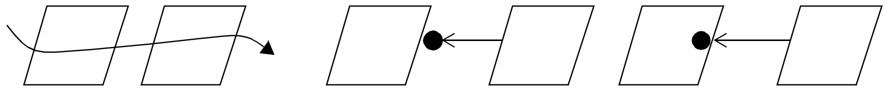
Some implementation technologies allow data to be transferred as part of an asynchronous interaction (for example, mailboxes), whereas others do not (for example, semaphores). Unless explicitly drawn as below, the default assumption would be that data transfer is possible. With semaphores, data may be transferred somewhat cumbersomely through a shared object, as in the example shown below (forgetting about the use of dynamic objects and slots for the moment, because they do not affect the issues being discussed here). This leaves the shared object open for access by both threads, requiring that it have mutual-exclusion protection.



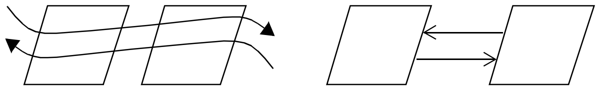
A synchronous connection puts control of progress along the path in the hands of the second thread. Assuming the connection is used only momentarily to transfer data, this should not slow up the first one too much but it may slow it up somewhat. Care has to be taken not to "procedurize" the second thread by locking the two together while it performs its responsibilities along the path. Otherwise, why have more than one thread?



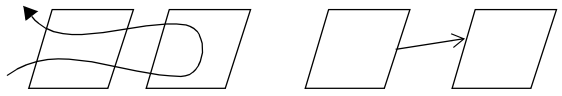
Another possibility is to use a synchronous connection in the opposite direction to the path. This puts control of progress along the path back in the hands of the first thread. The send-blocked or reply-blocked notation indicates that it may have to wait if the first thread has nothing for it.



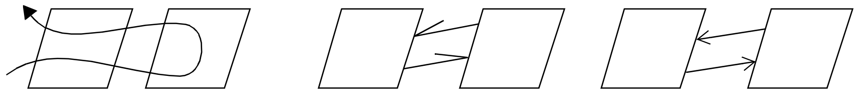
An interesting case is where there are paths in both directions and we innocently use the same connection pattern in both directions, for example, as shown below for a synchronous pattern. Connection patterns like this can cause deadlock. This is an example of needing to see more than one path to make design decisions about connections.



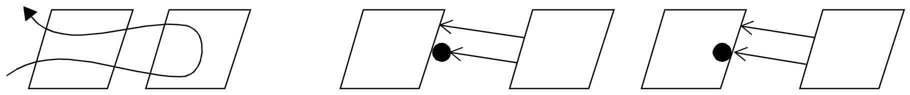
Excursions The case of an *excursion* requires two-way interaction, which may be accomplished in many ways. The most obvious way is to use a synchronous connection in the direction of the excursion. However, this may procedurize the first thread in an unintended fashion.



An excursion may be implemented with bidirectional synchronous or asynchronous interactions, one in the forward direction to signal its start and one in the reverse direction to signal its end. This avoids procedurization.



Another way of avoiding procedurization is to implement the forward path of the excursion with a send-blocked or reply-blocked synchronous interaction in the opposite direction to the path, requiring the destination of the excursion to anticipate it. The closing of the excursion loop would be accomplished by a second interaction in the direction of the return path (shown synchronous here, but it could be asynchronous).

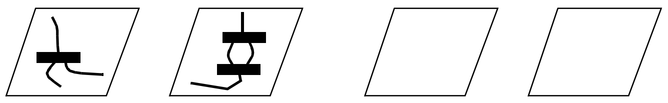


Deciding between these alternatives will likely depend on issues other than just single paths.

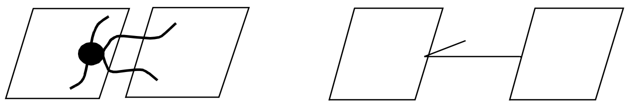
9.3.2 Path Forks, Joins and Waiting Places

This section summarizes patterns for various positionings of threads in relation to forks, joins and waiting places in paths.

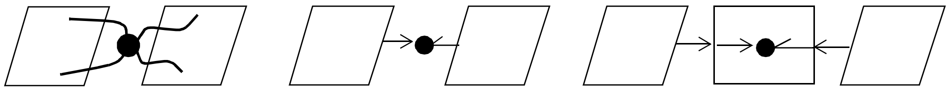
AND forks and fork-joins wholly inside threads imply don't-care ordering. There is no implication for collaboration graphs. The code of the thread may interleave the responsibilities of the paths in any arbitrary fashion.



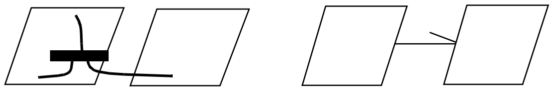
Asynchronous coupling between paths inside a thread calls for an asynchronous connection, with the triggering thread the initiator and the other thread the responder. The connection may be one with either external or internal waiting. If there is timeout (not shown), the internal-waiting solution below places responsibility for it directly in the responder thread.



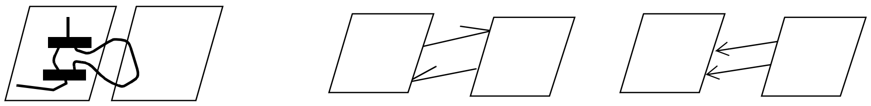
Asynchronous coupling between paths may be *between* threads instead of inside them, implying some sharing of responsibility. This suggests an asynchronous connection with external waiting. The coupling may be positioned inside an intermediate team or object, in which case its operational nature is hidden from the threads.



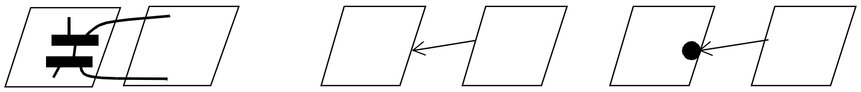
Fork concurrency as below calls for an asynchronous connection. Since there is no consequent join in this map fragment, no other connections are needed.



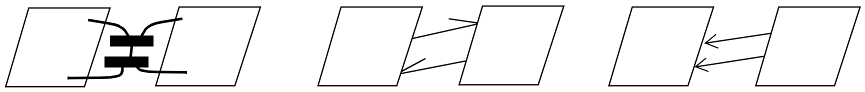
Fork-join concurrency as below calls for a nonprocedurizing pattern.



Private synchronous coupling is the only case that demands procedurizing.



Shared synchronous coupling may require several connections, including one or more connection-level rendezvous.



With these examples in mind to give guidance, the reader should be able to devise suitable connection patterns for other cases.

9.4 SUMMARY

In this chapter we:

- Identified unique characteristics of the collaboration graph notation, namely that it is both simple and broadly applicable to real time and object-oriented design.
- Gave an overview of the collaboration graph notation.
- Explained fine points of the notation relating to teams, structural dynamics, threads and ISRs.
- Provided guidelines on how to proceed from local patterns in use case maps to local ones in collaboration graphs.