A Method for the Specification, Composition, and Testing of Distributed Object Systems

Thesis by
Paolo A.G. Sivilotti

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1998
(Submitted December 1, 1997)

“O frati”, dissi, “che per cento milia
perigli siete giunti a l’occidente,
a questa tanto picciola vigilia

d’i nostri sensi ch’è del rimanente
non vogliate negar l’esperïenza,
di retro al sol, del mondo sanza gente.

Considerate la vostra semenza:
fatti non foste a viver come bruti,
ma per seguir virtute e canoscenza”.

Dante, *Inferno* XXVI, 112–120

Acknowledgements

Many people contributed to this research, and I am thankful to them all. I have been fortunate to work with Mani Chandy, my academic advisor. His ability to extract the elegant solution hidden behind the ugly hack, or the clever insight camouflaged as a mundane question has been an inspiration. Many thanks also to the other members of my thesis committee, Yaser Abu-Mostafa, Jim Arvo, Rajive Bagrodia, and Alain Martin, for taking the time to review this thesis and to make helpful suggestions.

Thanks to the other graduate students of our research group, past and present: John Thornley, Berna Massingill, Adam Rifkin, Eve Schooler, Rajit Manohar, Rustan Leino, Pete Carlin, Joe Kiniry, and Dan Zimmerman. Their individual influences shaped my research environment here at Caltech. Thanks also to Diane Goodfellow for her administrative support.

Thanks to the friends who helped make the good times fun, and the less good times bearable. Especially John Thornley, for many interesting and entertaining discussions (some even related to research!), and Mike Palmer, for being a beacon of sanity in a crazy world.

To Linda I owe a debt of gratitude, for her kindness and understanding. She has been my closest confidant, telling me what I needed to hear and listening to what I had to say.

Finally, thanks to my family — in particular, my parents, for their unwavering love and support; Mass and Ruth, for making sure I always had a home; Marco, for encouraging my academic pursuits; and Katie, who has asked for so little and has given me so much.

This research has been funded in part by an IBM Computer Science Fellowship, an NSERC '67 Fellowship, DARPA grant N00014-91-J-4014, the NSF under Cooperative Agreement No. CCR-9120008, and AFOSR grant F49620-94-1-0244.

Abstract

The formation of a distributed system from a collection of individual components requires the ability for components to exchange syntactically well-formed messages. Several technologies exist that provide this fundamental functionality, as well as the ability to locate components dynamically based on syntactic requirements. The formation of a correct distributed system requires, in addition, that these interactions between components be semantically well-formed. The method presented in this thesis is intended to assist in the development of correct distributed systems.

We present a specification methodology based on three fundamental operators from temporal logic: **initially** , **next** , and **transient** . From these operators we derive a collection of higher-level operators that are used for component specification. The novel aspect of our specification methodology is that we require that these operators be used in the following restricted manner:

- A specification statement can refer only to properties that are local to a single component.

- A single component must be able to guarantee unilaterally the validity of the specification statement for any distributed system of which it is a part.

Specification statements that conform to these two restrictions we call *certificates*.

The first restriction is motivated by our desire for these component specifications to be testable in a relatively efficient manner. In fact, we describe a set of simplified certificates that can be translated into a testing harness by a simple parser with very little programmer intervention. The second restriction is motivated by our desire for a simple theory of composition: If a certificate is a property of a component, that certificate is also a property of any system containing that component.

Another novel aspect of our methodology is the introduction of a new temporal operator that combines both safety and progress properties. The concept underlying this operator

has been used implicitly before, but by extracting this concept into a first-class operator, we are able to prove several new theorems about such properties. We demonstrate the utility of this operator and of our theorems by using them to simplify several proofs.

The restrictions imposed on certificates are severe. Although they have pleasing consequences as described above, they can also lead to lengthy proofs of system properties that are not simple conjunctions. To compensate for this difficulty, we introduce collections of certificates that we call *services*. Services facilitate proof reuse by encapsulating common component interactions used to establish various system properties.

We experiment with our methodology by applying it to several extended examples. These experiments illustrate the utility of our approach and convince us of the practicality of component-based distributed system development. This thesis addresses three parts of the development cycle for distributed object systems: (i) the specification of systems and components, (ii) the compositional reasoning used to verify that a collection of components satisfy a system specification, and (iii) the validation of component implementations.

Contents

List of Figures

List of Programs

Chapter 1

Introduction

1.1 Background

Distributed programs are parallel programs with physically distributed components. Each component can be viewed as an independent thread of control that can operate on local data, can communicate with other components, and can suspend and resume execution. Distributed components do not share state; the local data of one component is not visible to other components. Distributed solutions are appropriate when the system must interact with physically distributed computational entities. For example, components may be bound to remote sensors and actuators in a dynamic control application, to human engineers in a collaborative application, or to specific processor, memory, and display resources in a scientific application.

Distributed systems are becoming more prevalent. With the proliferation of workstations and of networks of these machines, there is an established hardware base for these applications. By bridging large physical distances between users, networked computers are a significant communication resource. The explosion of the World Wide Web [6, 5] has been dramatic evidence of the potential in exploiting networks. In addition to information exchange, there is also considerable interest in distributed collaborative, commercial, and entertainment applications. Distributed programming is poised to enter the mainstream of commodity software.

One methodology that is useful for managing the complexity of distributed systems is object orientation [12, 67]. This is a technique of program design that focuses attention on the data being manipulated [86, 91]. Object-oriented languages (such as C++ [96] and Oberon-2 [74]) contain sophisticated mechanisms for supporting user-defined data types, known as classes. One of these mechanisms, encapsulation, provides the ability to collect data and functionality within logical units, and to separate the implementation of these units from their interface. A class interface is a specification for how instances of the class (*i.e.*, objects) can be manipulated. Thus, an interface is an abstract representation of the contained data. This support for user-defined data types is one of the key characteristics that make object-oriented languages helpful for building complex systems, such as distributed ones. In a distributed object-oriented system, an object can be used to represent a single component of the system.

Commercial object-based frameworks for developing distributed systems typically use class declarations to verify a syntactic consistency between components. A class declaration of one component is used to verify that the component responds to a particular message (or method invocation). In recent years, an industry-wide standard, known as CORBA [77], has emerged for object-based distributed system construction. This standard overcomes many of the difficulties of connecting components in a heterogeneous environment by permitting them to communicate in a universal language.

1.2 Motivation

Even with the help of standards and object-oriented techniques for managing complexity, writing correct distributed programs remains a difficult task. Formal verification of sequential programs benefits from a well-understood principle of compositionality. This principle allows subprograms to be verified independently. Even with this advantage, however, rigorous application of formal methods in large sequential programs is prohibitively expensive. For distributed programs, the problem is even harder, since the rules for composition are more complicated. Reasoning about an individual component requires considering its interactions with the other components in the system. As a result, rigorous proofs of distributed programs can be complex and time consuming. At the same time, the complexity of these interactions makes the informal development of distributed systems more error prone than that of sequential systems. Therefore, programmers of distributed systems can benefit greatly from the use of formal arguments in reasoning about the correctness of their applications. The challenge is to make these techniques of formal verification practical for the mainstream development of distributed systems.

In this thesis, we explore a practical method for supporting the development of correct distributed programs. We propose a methodology for component specification based on a collection of semantic constructs. These semantic constructs are used to express properties of individual components, which can then be composed to derive properties of the system as a whole. Our methodology can be viewed as an extension to the object-based approach of many commercial frameworks for distributed system development. The CORBA standard (and its compliant implementations) provides a practical vehicle for our methodology, although its applicability is not restricted to this context. The theoretical contributions of this thesis are motivated by the desire for practical utility, while the pragmatic contributions are motivated by the desire for logical consistency.

1.3 Our Semantic Constructs

We define three fundamental constructs for the specification of component behavior. The first is the **initially** operator. This operator describes the set of possible initial states for the system. The second is the **next** operator. This operator describes, given a set of system states, the set of possible system states after the execution of a single action. The third is the **transient** operator. This operator describes a predicate on system states that, if true, is guaranteed to be false eventually. Together, these three operators form the basis of our specification language. From these basic building blocks, we derive a collection of higher level (and perhaps more familiar) operators such as **stable** and $\leadsto$ (pronounced "leads-to"). In addition, we derive a new operator, **follows** , well-suited to the description of systems with monotonic behavior.

We make use of these specification constructs in a restricted manner:

- A specification statement can refer only to state that is local to a single component.
- A single component must be able to guarantee unilaterally the validity of the specification statement for any distributed system of which it is a part.

Specification statements that conform to these two restrictions we call *certificates*.

The requirement that certificates refer only to state local to a single component has a significant consequence: A certificate can be tested within the context of a single component. Because components in a distributed system do not share state, testing general system properties at run-time requires gathering information from multiple address spaces. Also, careful attention must be given to the protocol used to gather this information in order to preserve its consistency. The locality of certificates, however, means that these properties can be tested and monitored simply by an examination of local state. In fact, we will see how certificates can be used to automatically generate the code required to test and monitor a component implementation.

The requirement that certificates be unilaterally guaranteed by a component also has a significant consequence: If a component is verified to implement a certificate, this verification argument applies regardless of the system of which the component is a part. That is, the certificate is a property of any system that contains the corresponding component. Thus, one property of any system is the conjunction of the certificates of the constituent components. This promotes proof reuse, an essential element of making formal methods practical in the marketplace.

Many interesting system properties, however, are not simply conjunctions of local component properties (*i.e.*, disjunction may be required). The maintenance of these system properties requires the coordination of multiple components. Components exchange messages to achieve a system goal or maintain a system invariant. We introduce the notion of a *service* as a reusable abstraction of a coordination paradigm. Two services are presented in detail: tokens (indivisible units) and logical clocks (monotonic counters and time stamps of messages). This is not an exhaustive list; it is meant to illustrate how paradigms of component interaction can be captured in services and how the specification and use of these services is integrated in our methodology.

1.4 Contributions

The contribution of this thesis is four-fold:

1. A new specification methodology, based on certificates, is introduced. This methodology is based on a collection of well-known operators, but adds some restrictions on their use. These restrictions result in two advantages: testability and a simple rule for composition. Several extended examples are presented as an exploration of the practical utility of this methodology.

2. A new temporal operator, **follows**, is defined. This operator combines both safety and progress properties. It can be used to succinctly describe and reason about systems with monotonic state. Several theorems are given for manipulating this operator, and examples are presented illustrating the conciseness of proofs based on this operator.

3. The use of certificates for testing and debugging component implementations is examined. We present a characterization of a restricted class of certificates that permits the automatic generation of a testing harness for component implementations. This harness, consisting of assertions and trace recordings, can detect the violation of certain

component properties and can warn of the possible violation of others. The capacity to generate this testing harness automatically is therefore a great benefit to the programmer debugging an implementation.

4. The abstraction of component coordination paradigms as services is explored. Two services (tokens and logical clocks) are specified and the corresponding component specifications given. Services represent basic protocols whereby a collection of components maintains or establishes a system property.

1.5 A Simple Example: The Shared Queue

This section presents a small example illustrating our approach to component specification. This example is intended to give the reader a flavor for the specification language presented in the thesis. The simplicity of this example permits the application of many different specification methodologies apart from our own.

Consider a component designed to store a queue of integers. This queue is shared among other components in the system. These components may add an integer to the queue or request the deletion of the integer at the head of the queue. For this simple example, we do not address the use of the values in the queue.

The declaration of this component in a CORBA-compliant system is given in Program 1.1. This declaration is used by the underlying run-time system to provide a variety of crucial functionalities; for example, the type-checking of procedure calls to remote objects, the marshaling of arguments for transmission over the network, and the instantiation or retrieval of the target object when a remote request is received.

```
interface Queue  {
  oneway void add (in long i);    //add an element to the (tail of) queue
  oneway void remove ();          //delete the element at the head of queue
};
```

Program 1.1: A CORBA declaration of a queue object.

This declaration does not, however, provide any information about the behavior of the component. The manner in which this component can be used must be inferred from an informal English description and the names chosen for the operations. The inference of semantics from informal descriptions is dangerous, as such descriptions can be vague or ambiguous. For example, it is not clear what effect a `remove()` operation has on a `Queue` component that contains no elements. Is it simply ignored, or is it stored as a request to delete an element as soon as one is available for deletion?

We extend the syntactic component declaration given above with our semantic constructs. An example of a component description for `Queue` is given in Program 1.2. This description relates the state of the component (as stored in its local variable) to the operations that are performed. The first certificate defines the length of the queue (`|Q|`), given the number of `add()` and `remove()` operations that have been performed (`deln(add)` and `deln(remove)`, respectively). The second certificate defines the value of each element of the queue (`Q[i]`) given the sequence of elements added (`D(add)[]`) and the number of `remove()` operations that have been performed (`deln(remove)`). Notice that these certificates refer only to local values and that their validity can be unilaterally guaranteed by a correct implementation.

```
Component Queue  {
  local vars :=  Q : sequence of int      //Queue of integers
  rpc targets := add(int) (unordered)
                 remove() (unordered)
  certificates :=
    Invariant.(|Q| = MAX(deln(add) - deln(remove), 0))
    Invariant.(Ai 0 <= i < |Q| : Q[i] = D(add)[i+deln(remove)])
}
```

Program 1.2: Component description for a shared queue.

The relationship of these values is graphically presented in Figure 1.1. This certificate-based description eliminates any ambiguity concerning the behavior of the `Queue` component. A `remove()` operation is never ignored. If the queue is empty, the next element added will be immediately discarded.

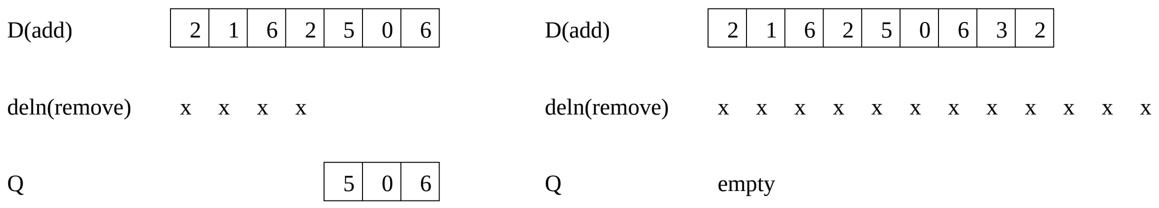

Figure 1.1: The state of the queue is determined by the sequence of `add()` operations and the number of `remove()` operations.

The component description for a queue that disregards (or "drops") requests to remove an element when it is empty is given in Program 1.3. The external interface for this `QueueDrop` component is the same as that for the `Queue` component. The difference is in the local state maintained by these components (the latter maintains a count of the number of elements removed from the queue), and in the component specification as given by the certificates. The first three certificates for the `QueueDrop` component define the value of the variable `removals`. The **initially** certificate specifies that the initial value of `removals` is 0. The **next** properties that follow require, respectively: that while the queue is empty, the value of `removals` be constant, and that while the queue is nonempty, its value increase as `remove()` requests are received. The last two certificates define the length of the queue and the values of the elements in the queue, given the sequence of `add()` operations and the value of the `removals` variable.

```
Component QueueDrop  {
  local vars :=  Q : sequence of int       //Queue of integers
                 removals : int            //number of successful removals
  rpc targets := add(int) (unordered)
                 remove() (unordered)
  certificates :=
    Initially.(removals = 0)
    (removals = k ^ |Q| = 0)  Next  (removals = k)
    (removals = deln(remove) - k ^ |Q| > 0)  Next  (removals = deln(remove) - k)
    Invariant.(|Q| = deln(add) - removals)
    Invariant.(Ai 0 <= i < |Q| : Q[i] = D(add)[i+removals])
}
```

Program 1.3: Alternate component description for a shared queue.

This small example is meant to illustrate the nature of component specification through the use of our certificates. These certificates can be mapped to a testing and debugging infrastructure. This mapping is presented in Chapter 5. These specifications can also be used in compositional proofs to establish properties of systems built up from these components. Extended examples of such proofs are given in Chapters 7 and 8.

1.6 Design Decisions

There are many choices made in the design of the methodology we present. Different choices could have been made, resulting in a different methodology, perhaps applicable to different models of computation. The principal design decisions are outlined below together with a short discussion of the trade-offs involved.

Asynchronous Messages. All messages in our model of computation are asynchronous. That is, a component continues to execute after sending a message. In systems with globally distributed components, the round trip communication delays associated with synchronous messages can be significant. For these systems, we expect asynchronous semantics to be the norm. The choice of asynchronous messaging also simplifies our model of execution by permitting us to treat methods as atomic. A component responds to the delivery of a message by possibly changing local state and by possibly sending one or more messages. On the other hand, there are cases for which synchronous semantics are more convenient (*e.g.*, for a method that returns data to the invoker). In these cases, the synchrony can be simulated with a call-back protocol.

The Role of Time. We do not explicitly include the physical passage of time in our model of computation. Our methodology does not permit the expression of real-time constraints. One can assert that "action X will occur eventually", but not, for example, that "action X will occur in the next two minutes". This approach is commonly taken in temporal logics and has the advantage of simplifying component refinement. Furthermore, most networks used for noncritical systems do not provide real-time guarantees for message delivery. On the other hand, a specification augmented with time guarantees would indicate to a client how long to wait before determining that a broken component is unresponsive.

Fault-Free Channels. We consider only channels that deliver messages without loss, duplication, or corruption. There is no bound on the length of time required to deliver a message, but the message is eventually delivered. Also, buffers for incoming messages are assumed to be infinite. These assumptions simplify the presentation by obviating arguments concerning faulty delivery.

Message Reception. Our model does not contain an explicit buffer of delivered, but unreceived, messages. That is, a component cannot probe to determine whether a particular message has been delivered and cannot selectively choose which messages to receive. This model is inspired by one-way remote procedure calls. If the ability to process messages in a particular order is required, a component can implement and maintain a mailbox into which all delivered messages are placed, and from which the component selects the next message to process.

1.7 Thesis Structure

In Chapter 2, we define our model of computation. The notions of system state, component actions, and computations are formalized. The message-passing layer whereby components interact is described informally. The CORBA standard for distributed object-based system development is discussed briefly. This commercial standard provides a practical context for the specification methodology we propose. This chapter also introduces the notation that will be used in the remainder of the thesis.

In Chapter 3, we describe the fundamental elements for component specification: certificates. The fundamental certificates (**initially** , **next** , and **transient**) are defined and a collection of higher level operators derived. These certificates are then used to formalize the channel properties of the message passing layer, along with some corollaries of these properties. Finally, we present and characterize some special cases, which we call *simple certificates*, of these fundamental certificates. Finally, some special cases of these fundamental certificates are presented and characterized. This subset of certificates we call *simple certificates.*

In Chapter 4, we examine more closely the **follows** operator introduced in the previous chapter. This is a new operator that combines both safety and progress properties in a single expression. Some theorems for the manipulation of expressions involving **follows** are presented and proven. As an illustration of the use of this operator, a succinct proof of a canonical distributed problem (the earliest meeting time problem) is given. To further illustrate the utility of **follows** , we use the theorems of this chapter to prove the corollaries of the channel properties given in the previous chapter.

In Chapter 5, we describe how certificate-based specifications can be used for the testing and debugging of component implementations. For each fundamental certificate, the framework for its translation into a collection of auxiliary variables and run-time assertions is given. Simple certificates enjoy an automatic mapping into a complete testing and run-time warning harness for a component. For properties that can be tested (*i.e.*, safety properties), this harness is used to generate an exception when the property is violated. For properties that cannot be tested (*i.e.*, progress properties), this harness is used to generate run-time warnings and traces that are useful for debugging when a component is suspected of being erroneous.

In Chapter 6, we introduce services. Services are frequently-used paradigms for component interaction and coordination. Two services are presented in detail: tokens and logical clocks. Each represents a system invariant that cannot be unilaterally guaranteed by a single component, and hence cannot be captured by a single component's certificates. An extended example is given that illustrates the use of services (in particular, tokens) in the development of a larger system.

In Chapters 7 and 8, we put our specification methodology to the test, using it to develop two extended examples. The first is a distributed auction, in which components submit bids until a winner and price are established. The second is a distributed branch and bound tree search, in which components are given subtrees to search as well as periodic updates on the best bound found. These chapters illustrate both the certificate-based specification of individual components as well as the compositional proofs of these specifications to establish properties of the entire system.

In Chapter 9, we outline some related work. Our research is compared with some of the other component-based efforts towards correct distributed system construction.

In Chapter 10, we conclude with a summary of our findings and a discussion of the utility of our approach. Future research directions are also outlined.

Chapter 2

Model of Computation

In this chapter, we define our model of computation. Components and the message-passing layer through which they interact are presented operationally. The notions of system state, a component action, and a computation are presented. A commercial instance of this model, the CORBA standard for distributed object-based system development, is discussed. This commercial standard provides a practical context for the specification methodology we propose. This chapter also introduces the notation and notational conventions that will be used in the remainder of the thesis.

2.1 Operational Overview of Model

In this section, we give an informal, operational overview of our model of computation. The model is quite general and is satisfied by a variety of commercially available systems. The goal of this section is to introduce some of the terminology and give some context for the kinds of systems addressed in this thesis.

2.1.1 Component Implementation

A distributed system consists of a collection of components. Components are implemented by programs that each reside in a single address space, have local state, and have a thread of control. We define an *implementation object* to be such a program. That is, an implementation object is a component that resides in a single address space.

An implementation object is an object in the traditional sense: It can be seen as a collection of data and functions (called *methods*) that manipulate this data. One of these methods is designated as the *constructor*. The constructor is executed when the implemen-

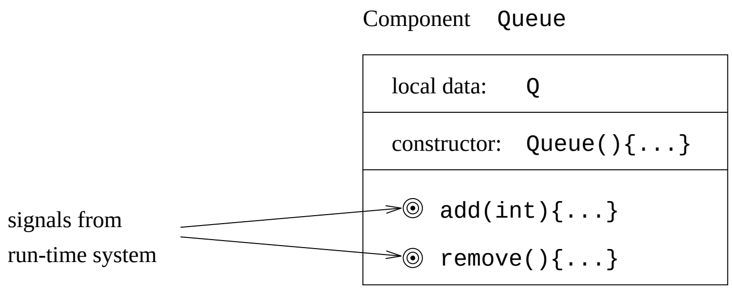

Figure 2.1: Each method is a target for signals from the run-time system.

tation object is first instantiated. The other methods are executed as a result of signals generated by the run-time system. They are identified by a function name and a list of argument types.

For example, consider the component discussed in Section 1.5 of the previous chapter that implements a queue. This component has local data (the variable `Q`) and three methods: a constructor, a method to enqueue an element (`add(int)`), and a method to dequeue an element (`remove()`). The last two methods are executed in response to signals from the run-time system. In Figure 2.1, each is depicted as a target for these signals.

An implementation object is always in one of two states: active or idle. Initially, it is active. An active implementation object has an executing thread of control and can perform the following operations:

1. It can change any element of its local state (including becoming idle).

2. It can send messages to components in the system. A message is a request for the target implementation object to execute one of its methods.

The local state of an implementation object includes the values of all its variables, the history of messages that it has sent and of messages that have been delivered to it, as well as whether it is active or idle.

An idle implementation object, on the other hand, does nothing except wait for a signal. A signal is generated either by the delivery of a message or by the run-time system in response to some stimulus by the environment (*e.g.*, an instrument detects a certain critical pressure, or a countdown timer expires). A signal causes an idle implementation object to become active and begin executing the method specified by the signal.

In our simple model, only an idle implementation object can detect a signal. An active one cannot probe the run-time system to discover if a signal is pending.

2.1.2 Component Interaction: The Message-Passing Layer

Each method of a component corresponds to a target for signals by the run-time system, as described above. We call these targets *RPC targets*, since they correspond to an interface for remote procedure calls (RPC's). These RPC targets are the mechanism whereby one component can invoke methods of another. A component invokes other components' methods by sending a message that specifies: (i) which method is to be invoked, (ii) the arguments to this method, and (iii) the identity of the component sending the message. The message-passing layer is responsible for transporting this message to its destination and signaling the appropriate RPC target.

For example, if `Storage` is a reference to an instance of the `Queue` component described above, an RPC invocation in an implementation language such as C++ might take the form:

```
Storage->add(2);
```

We consider the identity of the sender to be implicitly part of the invocation.

Component methods do not have a return value. The action of invoking an RPC is nonblocking. The message is sent and the sending component continues execution immediately. A method with a return value can be implemented with a call-back protocol. The invocation of the method includes the identity of the invoker. When the method completes, a message is sent to the invoker containing the result. By waiting for this result to be returned, the invoker achieves the effect of synchronous semantics. In order to present the simplest model with general applicability, we do not include synchronous semantics in our model directly.

The messages exchanged by components can be viewed as either oneway RPC requests or as messages on asynchronous channels with infinite slack. The two models are equivalent, and we will use the associated terminology interchangeably.

Channels are guaranteed to deliver messages without loss, duplication, or corruption. This delivery is guaranteed to occur in some finite, but unbounded, time. We do not specify as part of the model whether or not messages are delivered in the order sent. Some implementations of the message-passing layer may provide ordered delivery, others may not. We refer to this characterization of the message-passing layer as the *delivery discipline*. Other delivery disciplines are possible, such as faulty or noisy channels, but are not addressed here.

We associate a channel with a history of actions: a sequence D of deliveries and a sequence S of sends. We will also use the following functions on these histories:

- *delp* and *sentp*. The former (latter) predicate is true exactly when the history contains a delivery (send) action.

- *deln* and *sentn*. The former (latter) integer function is the number of delivery (send) actions in the history.

- *del* and *sent*. The former (latter) function returns the value of the last message delivered (sent). To refer to a particular delivery (send) action in the history, the sequence D (S) is subscripted. For example, $D[\mathit{0}]$ is the value of the first message delivered, and $S[\mathit{sentn} - \mathit{1}]$ is the value of the last message sent. When there has not been a delivery (send) action, the value of *del* (*sent*) is still defined; it is an arbitrary value chosen from the message type of the channel.

- *delm*. This function returns the value of the greatest message delivered. It is used in conjunction with monotonic channels (and hence the message type has an ordering relation).

A channel is uniquely identified by three things: (i) the identity of the sender, (ii) the identity of the receiver, and (iii) the name of the RPC target that is the destination. The application of the above functions, then, specifies these pieces of information. For example, for a component S and a component R with method m, $\mathit{deln}(S, R, m)$ is the number of messages that have been delivered on the channel from S to R at RPC target m. Often, these functions will be used in a context in which one or more of these parts of the channel identity can be understood implicitly from context, in which case that part will be omitted.

2.1.3 Component Descriptions

In an object-oriented programming language, an object is described by an interface declaration. Similarly, in our model of distributed systems, a component is described by what we call a *component description*. A component description contains the following information: (i) a declaration of constructed types used in declarations of local variables and constants as well as in arguments of the component's methods, (ii) a list of local constants, (iii) a list of local variables, (iv) a list of RPC targets, (v) a list of components to which this component can send messages, and (vi) a list of certificates. When a particular list is empty, the corresponding entry in the component description is omitted. Syntactically, a component description has the form given in Program 2.1.

We do not present a strict formalization of the syntax of component descriptions. The notation used is a mixture of some common structures from popular programming languages

```
Component <component name>  {
  local types :=  <declarations of constructed types>
  local const :=  <declarations of local constants>
  local vars :=   <declarations of local variables>
  rpc targets :=  <list of method signatures> <(delivery and sender disciplines)>
  neighbors :=    <list of components>
  certificates := <list of certificates>
}
```

Program 2.1: Syntax of a component description.

such as C++ and Pascal. For example, variables are declared in a Pascal style with a variable name followed by a colon and the type of the variable. Comments begin with "//" in the style of C++ comments.

The fundamental types are those of many programming languages: characters, integers, and floating point values. The constructed types are also familiar: structures, arrays, sequences, and sets of fundamental or constructed types. Components are themselves types. The first section of a component description allows these constructed types to be named. These names can then be used in the declaration of methods and local variables.

As an example, recall the component description given in Chapter 1 (and repeated here in Program 2.2). There are no local types in this description because no constructed type are named (*i.e.*, the only argument to a method is a fundamental type and the only local variable has type sequence of integers). This component has a single local variable, a sequence of integers called `Q`. Next, the RPC targets `add(int)` and `remove()` are listed. The `unordered` qualification that appears after each RPC target refers to the message delivery discipline that is assumed (*i.e.*, unordered but reliable delivery of messages). The certificates section contains properties that will be discussed in Chapter 3 (where the meanings of `Invariant`, `deln(add)`, `deln(remove)`, and `D(add)[i]` are all defined). The remaining symbols, such as `MAX`, the length of a sequence `|Q|`, and the universal quantification `(Ai : p.i)`, are assumed to be understood from common mathematical notation and usage.

2.2 State-Based Model

A fair interleaving semantics is used to represent the behavior of a distributed system. That is, a computation is defined to be an infinite sequence of actions. Each action in the sequence is an action performed by a component in the system. The fairness requirement is that if an action is enabled, eventually either the action appears in the trace (*i.e.*, is

```
Component Queue  {
  local vars :=  Q : sequence of int      //Queue of integers
  rpc targets := add(int) (unordered)
                 remove() (unordered)
  certificates :=
    Invariant.(|Q| = MAX(deln(add) - deln(remove), 0))
    Invariant.(Ai 0 <= i < |Q| : Q[i] = D(add)[i+deln(remove)])
}
```

Program 2.2: Component description for a shared queue.

allowed to execute) or it ceases to be enabled. Our model permits three kinds of actions: (i) a component can modify its local state, (ii) a component can send messages to other components, and (iii) the message-passing layer can deliver a message. We consider the delivery of a message to be atomic with the execution of the associated method. Similarly, we consider the entire execution of a method to be an atomic action. Hence, methods are required to terminate.

The state of the system, as defined above, is the state of the individual components and the state of the message-passing layer. For the message delivery disciplines described above, the state of the message-passing layer is functionally determined by the state of the individual components. The computation begins in some initial state where the message-passing layer has not delivered any messages and the component states are all determined by their initialization. Each action then maps the current state to a new state. A computation, therefore, can either be viewed as a sequence of actions or as the corresponding sequence of states.

As is common with such semantics, we permit stuttering [61, page 260]. That is, a system is allowed to do nothing for an arbitrary but finite number of steps.

2.3 An Instance of the Model: The CORBA Standard

This section presents a commercially available product that can be used as an implementation of our model as described above.

2.3.1 History

The Object Management Group (OMG) was formed in 1989 as a consortium of industries, universities, and research laboratories. Its membership currently numbers more than 750 software vendors, developers, and researchers, including: Apple Computer, Digital Equipment, Hewlett-Packard, International Business Machines, Microsoft, Netscape Communications, Novell, Silicon Graphics, and SunSoft. The mandate of this organization has been to promote the theory and practice of object technology for the development of distributed computing systems. One of the most significant contributions by the OMG has been a standard for distributed object-based systems, known as CORBA [77]. Introduced in 1991, the CORBA specification defined the middleware to support distributed object communication. Several implementations of this standard became available soon after. Adopted in 1995, a revision of the specification (CORBA 2.0) defined the requirements for interoperability between these implementations. Currently, many commercial implementations of the CORBA 2.0 standard are available, such as Orbix (by IONA), VisiBroker (by Visigenic), SOM/DSOM (by International Business Machines), ORB Plus (by Hewlett-Packard), and Object Broker (by Digital Equipment).

2.3.2 Architecture and the IDL

At the core of CORBA is an object request broker that intercepts the dispatch of a method invocation and is responsible for finding the target object that implements the method, marshaling and demarshaling the arguments, activating the method, and returning the result. In the language of the CORBA specification, the component issuing the method request is the *client*, while the destination component of the request is the *server*.

There are two mechanisms whereby this dispatch can occur. The first is via a dynamic invocation interface. This interface permits a client to construct a method invocation at run time, retrieving the method signature from an interface repository and appending the required arguments to a dynamic structure before issuing the request. The second mechanism is via stub and skeleton code that is statically generated from an object interface description. A graphical representation of this process is given in Figure 2.2.

Both approaches require the definition of implementation object interfaces (the former to create repositories of interface information and the latter to generate the stub and skeleton code). Part of the CORBA definition is a universal language for this purpose. This language, known as IDL (for Interface Definition Language), provides the syntax required to describe the signatures of methods that can be invoked on an object. As an example, recall the shared queue example used in Chapter 1. The IDL definition of this object is repeated here in Program 2.3.

All CORBA implementations include an IDL parser responsible for translating an IDL

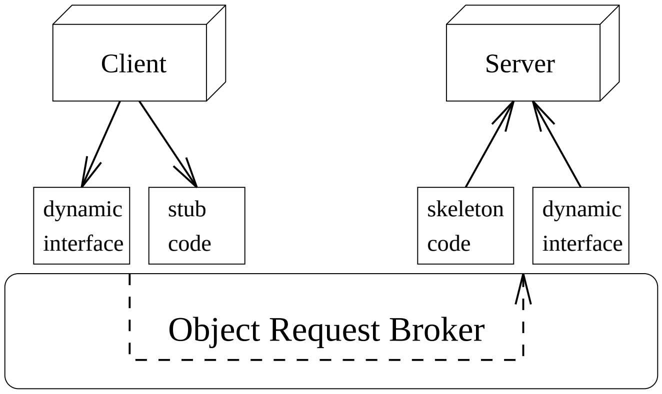

Figure 2.2: Method Invocation in CORBA.

```
interface Queue  {
  oneway void add (in long i);     //add an element to the (tail of) queue
  oneway void remove ();           //delete the element at the head of queue
};
```

Program 2.3: An IDL definition of the `Queue` interface.

definition into stub code and skeleton code. The stub code is a collection of functions called by the client to invoke a remote method. The invocation in the client code is then identical to an invocation of a local method, the only difference being that the target of the invocation is a remote object. The skeleton code is a framework that the implementor of the server program completes. In general, for each method in the IDL definition, a function is created by the IDL parser that takes the required arguments. It is the responsibility of the programmer to code the body of the function so as to produce the desired result.

For example, consider the IDL definition of `Queue` in Program 2.3. IBM's SOM/DSOM implementation of the CORBA standard produces the skeleton code given in Program 2.4 from this IDL definition. The parser adds some SOM-specific code and provides two function templates for the programmer to complete (one for `add()` and one for `remove`).

2.3.3 Other Implementations

Although CORBA provides a natural context for the methodology proposed in this thesis, our methodology is not confined to this context. Indeed, any implementation of the model for distributed systems presented in this chapter is suitable for the application of our

```
/*
 *  This file was generated by the SOM Compiler and Emitter Framework.
 *  Generated using:
 *      SOM Emitter emitxtm: 2.41
 */

#ifndef SOM_Module_queue_Source
#define SOM_Module_queue_Source
#endif
#define Queue_Class_Source

#include "queue.xih"

SOM_Scope void  SOMLINK add(Queue *somSelf,  Environment *ev,
                            long i)
{
    /* QueueData *somThis = QueueGetData(somSelf); */
    QueueMethodDebug("Queue","add");

}

SOM_Scope void  SOMLINK remove(Queue *somSelf,  Environment *ev)
{
    /* QueueData *somThis = QueueGetData(somSelf); */
    QueueMethodDebug("Queue","remove");

}
```

Program 2.4: SOM/DSOM code skeleton for `Queue`.

approach to component specification and composition. Other available implementations include Java [32] with RMI, CC++ [16, 93], Fortran M [27], PVM [30], and MPI [95] to name a few. To avoid confusion in the presentation, a single practical context is used in this thesis, namely the CORBA standard.

2.4 Notation and Conventions

2.4.1 Functions and Operators

The usual mathematical and logical operators are used. Function application is denoted with the infix "." operator. For example, the function f applied to x is written $f.x$. Since predicates are functions (from the state space to the booleans), a similar notation is used for predicates. For example, the predicate p applied to state s is written $p.s$. Function application associates to the left. For example, $\mathbf{stable}.p.C$ should be read as $(\mathbf{stable}.p).C$.

The usual binding powers are given to operators:

1. . (function application)
2. $\neg$
3. arithmetic operators (including $=$), with their usual binding power
4. $\wedge$, $\vee$
5. $\Rightarrow$, $\Leftarrow$
6. $\equiv$

The temporal operators introduced in Chapter 3 all have the lowest binding power (below $\equiv$).

2.4.2 Quantification

Quantification of a binary, associative, commutative operator op over a set of values is denoted $(\, op\; dummies \;:\; range \;:\; term \,)$, where $dummies$ is a list of dummy variables, $range$ is the range of quantification, and $term$ is the term of quantification. For example, the following formula is an expression that all even natural numbers satisfy the predicate p:

$$(\wedge n \;:\; n \in \mathbb{N} \wedge even.n \;:\; p.n \,)$$

The logical operators $\wedge$ and $\vee$, when used as quantification operators, will usually be written as $\forall$ and $\exists$ respectively. Similarly, $+$ will be written as Σ , in accordance with common usage. For example, the universal quantification of a predicate p over all even natural numbers given above will be denoted:

$$(\forall n \;:\; n \in \mathbb{N} \wedge even.n \;:\; p.n\;)$$

Often, the type of the dummy (*e.g.*, the natural numbers in the preceding examples) will be understood from context. If the entire range is understood from context, an abbreviated notation is used:

$$(\forall n \;::\; p.n\;)$$

Universal quantification over the state space will be denoted by the "everywhere brackets" as in [25]. For example, consider predicates $x > 10$ and $x > 5$ on a state space S that contains a variable x . The truth of the latter predicate is implied by the truth of the former. We write $[x > 10 \;\Rightarrow\; x > 5]$ as a shorthand for $(\forall s \;:\; s \in S \;:\; (x > 10 \;\Rightarrow\; x > 5).s\;)$.

We also adopt the convention that formulas with unbound dummies are implicitly universally quantified over these dummies. For example, the formula $\textbf{stable}.(x = k)$ where k is a dummy should be read as $(\forall k \;::\; \textbf{stable}.(x = k)\;)$.

2.4.3 Proof Format

Our proof format is based on [25]. It gives a justification (or hint) for each step. For example, the validity of $[p \;\Rightarrow\; q]$ might be established by a proof of the form:

$$\begin{array}{ll} & p \\ \equiv & \{\ \text{hint why } [p \;\equiv\; r]\ \} \\ & r \\ \Rightarrow & \{\ \text{hint why } [r \;\Rightarrow\; q]\ \} \\ & q \end{array}$$

Since our temporal operators **next** and $\leadsto$ introduced in Chapter 3 can be part of a weakening implication chain, we extend this notation to include these operators. For example, the property $(p \;\leadsto\; q).C$ might be established by a proof of the form:

$$\begin{array}{ll} & p \\ \Rightarrow & \{\ \text{hint why } [p \;\Rightarrow\; r]\ \} \\ & r \end{array}$$

$\rightsquigarrow$ $\{$ hint why $(r \rightsquigarrow q).C$ $\}$

q

Chapter 3

Certificates and Specification

In this chapter, we define our principal specification construct: the certificate. Certificates are used both to specify and to test individual components. Fundamental certificates are introduced and some examples are given illustrating how these certificates can specify various component properties. These certificates are also used to formally specify the channel properties in our model of communication. We then introduce simplifications based on restricted classes of certificates.

3.1 Component Specification

Recall that in our model, there are 3 types of actions: (i) a component can modify its local state, (ii) a component can send messages to other components, and (iii) the message-passing layer can deliver a message. As defined in Chapter 2, a computation is an infinite sequence of such actions [61, pages 10–12]. We identify a computation with the corresponding infinite sequence of system states.

A specification is a precise formal description of the behavior of a software system. We characterize the behavior of a system by the set of possible computations the system can generate. A specification, therefore, defines a set of permissible computations, and a system is said to implement a specification if every computation of the system is one of the computations permitted by the specification [61, Chapter 3]. We will use predicates on computations, known as *properties*, as specification statements.

Recall that components do not share state. This means that any action in the system can affect at most a single component's state. (Some actions — namely, communication actions — may also modify the state of the message-passing layer.) Consider the three types of actions defined in our model:

Read-write actions. Variables are local to a component, so the modification of a variable modifies only the state of that component.

Message sending. Sending a message modifies the state of the outgoing channel. The state of the component sending the message is also modified (*e.g.*, the history of messages sent is updated).

Message delivery. Similarly, the delivery of a message modifies the state of the incoming channel and the state of the component receiving the message.

Hence, a component's behavior can be defined in terms of its local state: the local variables, the messages it has sent, and the messages it has received.

We define a component *certificate* to be a specification statement that meets two restrictions:

1. The expressions contained in the statement involve only the local variables of the component and the messages sent by and delivered to that component.

2. The validity of the statement does not depend on the environment in which the component is placed.

Certificates enjoy a significant property: If a certificate is a property of a component, it is also a property of any system that contains that component. In the language of [20], certificates are examples of "exists-component" properties.

Despite the fact that specifications are meant to be visible to a component's environment, the presence of so-called local variables in these specification statements is not a cause for concern. These variables are elements of the component description (as defined in Chapter 2) and not part of the implementation. These variables are therefore specification (also known as "ghost," "auxiliary," or "thought") variables, and including them in the specification does not violate any principles of information hiding.

There are two parts to the specification of concurrent systems: safety and progress. An example of a safety property is that the value of variable `x` is always positive. A safety property can be violated by a finite sequence of actions. An example of a progress property is that eventually the value of variable `x` is greater than 5. Unlike a safety property, a progress property cannot be violated by a finite sequence of actions. We introduce two fundamental constructs for dealing with safety (**initially** and **next**) and one for progress (**transient**). From these basic concepts we will derive two more constructs for expressing safety (**stable** and **invariant**) and one more for progress (leads-to, written $\leadsto$). This formal system is known to be sound and relatively complete [20]. We will also derive a new construct (**follows**) that simultaneously expresses both safety and progress properties. We will then use these constructs to specify the channel properties of our model of computation and derive some useful corollaries.

3.2 Safety

3.2.1 Certificate initially

As defined in Chapter 2, a computation is a sequence of states. The first state is designated the initial state. The **initially** certificate is a predicate that holds in this initial state. That is, the components of the system have been instantiated and have initialized their local variables. For a predicate p and a system C, we define:

$$\textbf{initially}.p.C \triangleq (\forall\, computations < s_i \,:\, i \geq 0 > of\ C \,::\, p.s_0 \,)$$

Function application associates to the left, so we write $\textbf{initially}.p.C$ for $(\textbf{initially}.p).C$.

For example, consider a component `Philosopher` that cycles between three states: thinking, hungry, and eating. The current state is encoded in three boolean variables: `thinking`, `hungry`, and `eating`. When first instantiated, the component is in the thinking state. This component property is written:

$$\textbf{initially}.\mathit{thinking}$$

The component to which a certificate applies will typically be understood from context. Where there is opportunity for confusion, the component will be given explicitly. For example, the following expression is a boolean:

$$\textbf{initially}.\mathit{thinking}.\texttt{Philosopher}$$

3.2.2 Certificate next

Whereas the **initially** certificate describes the valid states at the beginning of a computation, the **next** certificate describes how a state in a computation constrains the state that follows it.

The specification statement p **next** q is a property of a component exactly when any state satisfying p is immediately followed in the computation by a state satisfying q.

$$(p\ \textbf{next}\ q).C \triangleq (\forall\, computations < s_i \,:\, i \geq 0 > of\ C \,::$$
$$(\forall\, i \,:\, i \geq 0 \,:\, p.s_i \Rightarrow q.s_{i+1} \,)\,)$$

We will refer to p and q as the prepredicate and postpredicate respectively.

Recall from our description of the computational model in Chapter 2 that the actions of the individual components are interleaved in a computation with actions of other components and of the underlying message-passing layer. Thus, a certificate involving **next** must allow for actions that do not change local state. So given p **next** q, it follows that p implies q:

$$(p \;\mathbf{next}\; q).C \Rightarrow [p \Rightarrow q] \tag{3.1}$$

The square brackets ("[ ]") are "everywhere brackets" [25] representing universal quantification over all states.

For example, consider the component `Philosopher` from the previous section that cycles between 3 states: thinking, hungry, and eating. To express that the component becomes thinking after eating, the following certificate is used:

$$\mathit{eating} \;\mathbf{next}\; \mathit{eating} \vee \mathit{thinking}$$

We define **next** to have a lower binding power than equivalence (and hence a lower binding power than disjunction).

Notice that this certificate does not ensure that the component will ever stop eating. It may be the case that the component continues to eat indefinitely. The above certificate says only that a state in which the component is eating is immediately followed by a state in which either the component is still eating, or the component is thinking. The property that the component will eventually stop eating is a progress property and is discussed in Section 3.3.

The **next** operator has appeared in several forms, including the **co** operator in [69], the *next* operator in [20], and the $\bigcirc$ operator in temporal logic [51, 99].

3.3 Progress

3.3.1 Certificate transient

A progress property states that something eventually happens. We employ a single fundamental certificate for expressing progress: **transient**. A transient predicate is a predicate that once true is guaranteed to be falsified eventually. The specification statement **transient**.p is a property of a component exactly when any state satisfying p is eventually followed in the computation by a state satisfying $\neg p$.

$$\textbf{transient}.p.C \triangleq (\forall\, computations < s_i \,:\, i \geq 0 > of\ C \,::\, (\forall\, i \,:\, i \geq 0 \,:\, p.s_i \Rightarrow (\exists\, j \,:\, j > i \,:\, \neg p.s_j\ )\ )\ )$$

For example, consider once again the component `Philosopher` used in previous sections. To express the fact that a `Philosopher` does not eat forever, we write the certificate:

$\textbf{transient}.eating$

3.3.2 Use of transient

On casual consideration, this property may appear to be of limited use as a certificate in our model of computation (recall that certificates are restricted to refer only to local variables and channels). We have postulated distributed components with methods whose invocation and execution occur as a single atomic action. That is, the action of the delivery of an RPC request by the message-passing layer and the execution of the corresponding method occur as a single atomic block. It seems, then, that any progress property will require the cooperation of more than one component, and hence cannot be unilaterally guaranteed by a single component.

It is important to remember, however, that there is not necessarily a one-to-one relationship between components and implementation objects. A single component may correspond to multiple implementation objects. It is the actions of the individual implementation objects that occur as atomic blocks. There are two important cases that give rise to multiple-object components: composition and clocks.

Case 1: Composition

Components are closed under parallel composition. That is, the parallel composition of components yields a new component. Even if the original components each correspond to a single implementation object, their composition does not. For example, consider two simple components, `Increment` and `Double`. The first component receives an RPC request containing an integer, increments the integer, then sends the result in an RPC request to a `Double` component. The second component receives an RPC request containing an integer, doubles the integer, then sends the result in an RPC request to a `Consumer` component. The descriptions of these components are summarized in Programs 3.1, 3.2, and 3.3.

The behavior of `Increment` is defined strictly in terms of safety properties. That is, there is a single atomic method, `inc()`. The delivery of a message to this RPC target is

```
Component Increment  {
  rpc targets := inc (int) (write-once)
  neighbors :=   d : Component Double
}
```

Program 3.1: Description of the `Increment` component.

```
Component Double  {
  rpc targets := dbl (int) (write-once)
  neighbors :=   c : Component Consumer
}
```

Program 3.2: Description of the `Double` component.

atomic with the calculation of the new value and the sending of this new value on to the neighbor `d`. A possible trace of execution of this component is represented graphically in Figure 3.1. Notice that every state is characterized by the same number of delivery and send actions, and by a particular mathematical relationship between these two sequences (the latter is an increment of the former). The behavior of `Double` is similarly defined strictly in terms of safety properties.

Now consider the composition of `Increment` and `Double` to form a new component. This composition binds the neighbor value `d` of the `Increment` component to a `Double` component. Pictorially, this composition is depicted in Figure 3.2. One of the properties of this new component is a progress property; namely, that the delivery of an RPC request `inc()` eventually results in a message being sent to the consumer. More formally,

$$\textbf{transient}.(delp(inc) \wedge \neg sentp(c, result))$$

Operationally, this temporal property is a result of the (nonatomic) communication that occurs between the `Increment` and `Double` components.

```
Component Consumer  {
  rpc targets := result (int) (write-once)
}
```

Program 3.3: Description of the `Consumer` component.

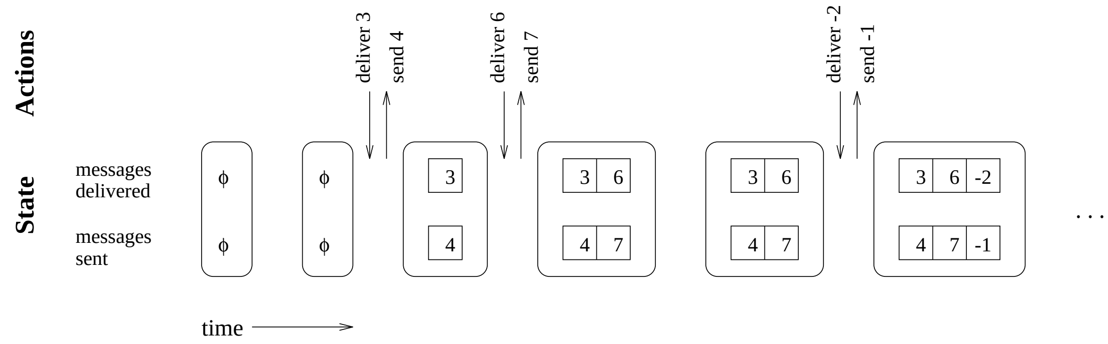

Figure 3.1: An example trace of execution for the `Increment` component.

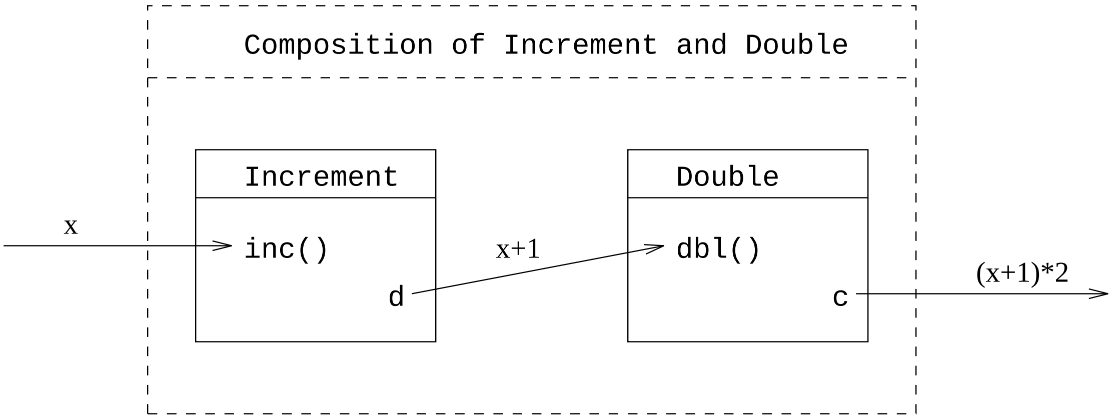

Figure 3.2: Composition of `Increment` and `Double` components.

Case 2: Clocks

The second case that gives rise to locally verifiable **transient** properties is the use of clocks by components. A clock is often used to generate a time-out signal when the component is interacting with the outside world (*e.g.*, a disk drive or a user). Such an environment may or may never provide the required input. To deal with this uncertainty, a component can use a time-out to terminate waiting for input that may never come.

For example, consider a distributed system that provides a simple two-way communication link for users. The link is established when a request by one user to initiate a session is accepted by the other user. The RPC targets involved in session initiation are shown in Program 3.4. The simple topology of this application is illustrated in Figure 3.3.

A component that has received a request for a session notifies the user and waits for a

```
Component Link  {
  rpc targets := request (unary) (ordered)
                 accept (unary) (ordered)
                 reject (unary) (ordered)
  neighbors :=   n : Component Link
}
```

Program 3.4: Description of the `Link` component.

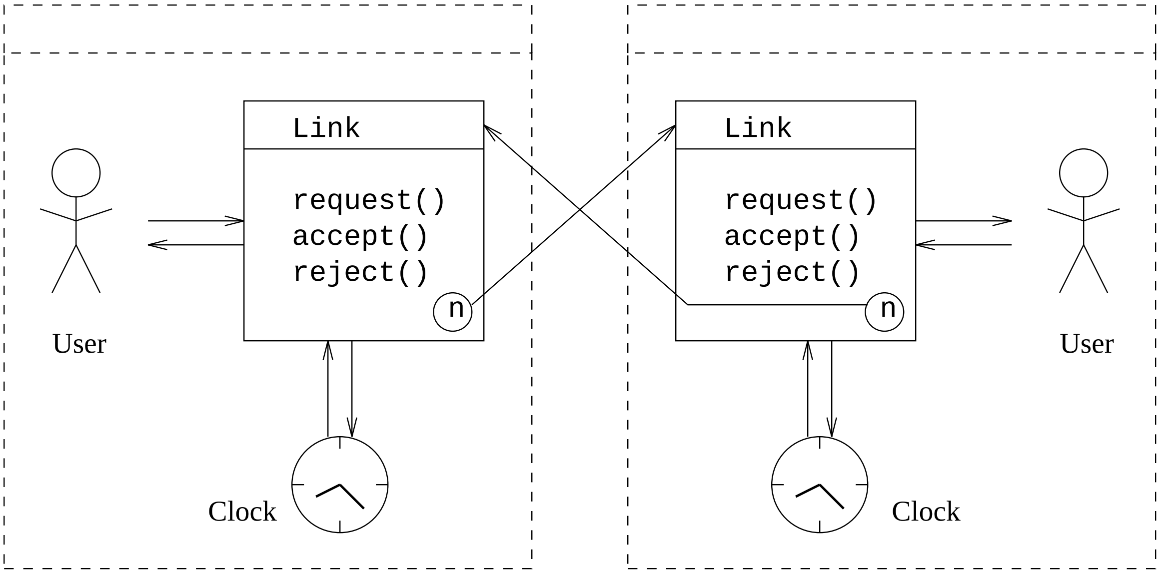

Figure 3.3: Topology of communication link between users.

decision whether to accept or reject the request. The user, however, may not be present and in this case the request should be rejected. The component must therefore guarantee that eventually a reply of some kind is sent. This guarantee is expressed by the following certificate:

$$\textbf{transient}.(\mathit{delp}(\mathit{request}) \wedge \neg(\mathit{sentp}(n, \mathit{accept}) \vee \mathit{sentp}(n, \mathit{reject})))$$

This guarantee can be implemented in any number of ways that provide a time-out signal.

Another way to illustrate the role of clocks is to consider a representation of a component's protocol. A common representation of a protocol for a communicating system is that of a finite state machine [43, 66, 70], where the transitions correspond to communication actions (*i.e.*, the sending and receiving of messages) [33]. Consider a typical client-server model, where a server object waits in a dormant state for an RPC request. When such a request is delivered, the appropriate method is executed (possibly resulting in several messages being sent) and the server returns to its dormant state. A protocol for such a system is represented in Figure 3.4. Sending a message is denoted with a minus sign, while the delivery of a message is denoted with a plus sign.

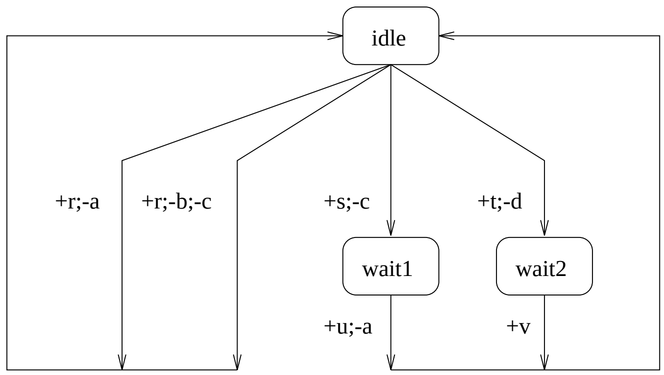

Figure 3.4: A server's protocol in a client-server system.

As a consequence of our model of computation, transitions from a state begin with the delivery of a message. These states (idle, wait1, and wait2) correspond to dormant states where the object is waiting for the delivery of an RPC request. The delivery may be followed by some number of send actions.

For a time-out, the component uses the clock to generate a signal. This signal is modeled as the delivery of a message. For example, the protocol for the `Link` component is given in Figure 3.5. In the state `wait2`, the component waits to receive a signal from either the user or the clock. If the user does not reply, the link request is rejected. Because the clock is trusted to generate a signal, the component can unilaterally guarantee that the state `wait2` is transient:

$$\textbf{transient}.\mathit{wait2}$$

3.4 Derived Certificates: stable, invariant, Leads-to, and follows

3.4.1 Certificate stable

A *stable predicate* is a predicate that once true remains true.

$$\textbf{stable}.p.C \quad \triangleq \quad (p \ \textbf{next} \ p).C$$

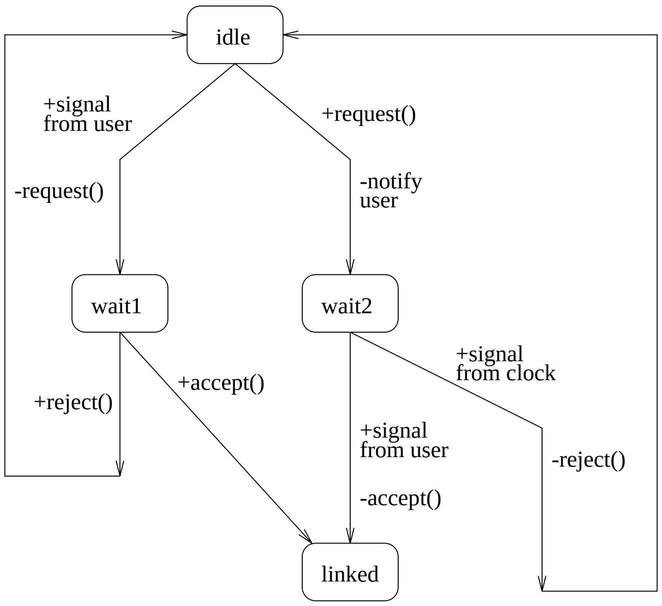

Figure 3.5: Protocol for the `Link` component.

An example of a **stable** property is that the system has terminated, or that a message has been delivered.

3.4.2 Certificate invariant

An *invariant predicate* is a stable predicate that is true initially.

$$\textbf{invariant}.p.C \quad \triangleq \quad \textbf{initially}.p.C \wedge \textbf{stable}.p.C$$

There is a subtle distinction between the notion of invariant as described above and the notion of "always true." In particular, a predicate that is always true in a computation may be too weak to be proven to be invariant using the above definition. The distinction centers around the consideration of reachable states and is eliminated by the adoption of a substitution axiom similar to that of UNITY logic [19, 87, 69].

An example of an **invariant** property is that a variable is always positive. Another example is that in the `Philosopher` component, exactly one of the booleans `thinking`, `hungry`, and `eating` is true:

$$\textbf{invariant}.((thinking \equiv hungry \equiv eating) \wedge (\neg thinking \vee \neg hungry \vee \neg eating))$$

3.4.3 Certificate Leads-to

A leads-to property, such as $p \leadsto q$, expresses the fact that once p becomes true q is or will be true [19, Section 3.4.5]. Leads-to can be defined in terms of transience and **next** [68].

A rule that we will use frequently in deriving leads-to properties is as follows:

$$\textbf{stable}.p.C \wedge \textbf{transient}.(p \wedge \neg q).C \Rightarrow (p \leadsto q).C$$

3.4.4 Certificate Follows

Consider two variables, `x` and `y` of the same type, and let this type have an ordering relation $\leq$ that defines a partial order (*i.e.*, the ordering relation is reflexive, transitive, and antisymmetric). The property x **follows** y expresses many things:

1. Both x and y are monotonically increasing.
2. The value of x does not exceed the value of y.
3. If the value of y exceeds some constant, then eventually the value of x will also exceed that constant.
4. The difference between x and y is an upper bound on how much x can increase in one step.

The **follows** operator is defined in terms of stability, invariance, leads-to, and a **next** property.

$$\begin{aligned}(x \textbf{ follows } y).C \quad \triangleq \quad & (\forall k :: \textbf{stable}.(x \geq k).C\ ) \wedge (\forall k :: \textbf{stable}.(y \geq k).C\ ) \\ & \wedge \textbf{invariant}.(x \leq y).C \wedge (\forall k :: (y \geq k \leadsto x \geq k).C\ ) \\ & \wedge (\forall k :: y \leq k \wedge x \leq k \textbf{ next } x \leq k\ )\end{aligned}$$

The ordering relationship used on x and y will typically be understood from context (*e.g.*, integers are ordered by less-than and predicates are ordered by implication). If there is a possibility for confusion, however, or if the **follows** property expresses a monotonically nonincreasing relationship, the ordering relation will be subscripted after **follows**; for example, x $\textbf{follows}_{\geq}$ y for integers x and y and x $\textbf{follows}_{\subseteq}$ y for sets x and y (the above definition is the definition of x $\textbf{follows}_{\leq}$ y).

The **follows** operator is interesting as it combines both safety and progress properties. It can be used to succinctly define the properties of some components and, in particular, channels (as will be seen in the next section). Some useful theorems for manipulating **follows** properties as well as some example proofs using **follows** are given in Chapter 4.

3.5 Channel Properties

Using the safety and progress primitives introduced in Sections 3.2, 3.3, and 3.4, we can now formalize the message-passing layer described operationally in the previous chapter. We associated a channel with a history of actions: a sequence D of deliveries and a sequence S of sends. We also introduced the following functions on these histories:

- *delp* and *sentp*. The former (latter) predicate is true exactly when the history contains a delivery (send) action.
- *deln* and *sentn*. The former (latter) integer function is the number of delivery (send) actions in the history.
- *del* and *sent*. The former (latter) function returns the value of the last message delivered (sent). To refer to a particular delivery (send) action in the history, the sequence D (S) is subscripted. For example, $D[\mathit{0}]$ is the value of the first message delivered, and $S[\mathit{sentn} - \mathit{1}]$ is the value of the last message sent. When there has not been a delivery (send) action, the value of *del* (*sent*) is still defined; it is an arbitrary value chosen from the message type of the channel.
- *delm*. This function returns the value of the greatest message delivered. It is used in conjunction with monotonic channels (and hence the message type has an ordering relation).

We now present the channel properties that will be used in our proofs. In the equations that follow, we will omit the argument (*i.e.*, the channel) to this function. First, the definitions given above are formalized. Next, the properties that depend on the message-delivery discipline (*i.e.*, ordered and unordered delivery) are given. Finally, the properties that depend on various sender disciplines (*i.e.*, write-once and monotonic sends) are given. The properties that follow from the message-delivery discipline are independent of what, if any, sender discipline is used. Similarly, the properties that follow from the sender discipline are based on the weakest message-delivery discipline (unordered delivery). These properties also hold with ordered delivery.

3.5.1 Definitions Common to All Channel Types

Notation. The notation $|X|$ is used to represent the number of elements in sequence X .

Definitions. We begin with the definitions of *deln* , *sentn* , *delp* , *sentp* , *del* , and *sent* .

$$deln = |D|$$

$$sentn = |S|$$

$$delp \equiv deln > 0$$

$$sentp \equiv sentn > 0$$

$$delp \Rightarrow del = D[deln - 1]$$

$$sentp \Rightarrow sent = S[sentn - 1]$$

The last two equations define the values of *del* and *sent* only when there has been an action in the corresponding history. We prefer to work with total functions, so we adopt an untyped view of our specification language, where undefined values are determined by Hilbert's "choice" operator [59] (representing an arbitrary value taken from the set of possible values). This approach is similar to the TLA specification language [56, 57]. We will take care not to reference these values in properties or proofs.

3.5.2 Properties that Depend on Message-Delivery Discipline

Unordered Channels. An unordered channel does not guarantee the preservation of the order of sent messages. It does guarantee, however, that messages arrive without loss or duplication.

Let $\sqsubseteq$ denote the ordering relationship between sequences corresponding to a subset relationship between multisets. That is, a sequence X is below ($\sqsubseteq$) a sequence Y if and only if every element X appears in Y with the same or greater multiplicity.

$$\begin{aligned} X \sqsubseteq Y \quad \triangleq \quad (\forall k :: \; & (\# i : 0 \leq i < |X| : X[i] = k) \\ & \leq (\# i : 0 \leq i < |Y| : Y[i] = k)) \end{aligned}$$

This message-delivery discipline is characterized by the following channel property:

$$D \ \textbf{follows}_{\sqsubseteq} \ S \tag{3.2}$$

From this property we can derive the following corollaries:

$$deln \ \textbf{follows} \ sentn \tag{3.3}$$

$$delp \ \textbf{follows} \ sentp \tag{3.4}$$

Ordered Channels. An ordered channel guarantees the preservation of the order of sent messages. Messages arrive without loss or duplication.

Let $\preceq$ denote the prefix ordering relationship between sequences. That is, a sequence X is below ($\preceq$) a sequence Y if and only if X is a prefix of Y .

$$X \preceq Y \quad \stackrel{\triangle}{=} \quad |X| \leq |Y| \wedge (\forall i \, : \, 0 \leq i < |X| \, : \, X[i] = Y[i] \,)$$

This message-delivery discipline is characterized by the following channel property:

$$D \ \textbf{follows}_{\preceq} \ S \tag{3.5}$$

This message-delivery discipline is stronger than unordered message delivery. That is, the following corollary can be derived from the channel property above:

$$D \ \textbf{follows}_{\sqsubseteq} \ S \tag{3.6}$$

Hence, the corollaries given for unordered channels are also corollaries for ordered channels.

In addition, the following corollary can be derived:

$$(\forall j \, : \, j \geq 0 \, : \, deln > j \wedge D[j] = k \ \textbf{follows} \ sentn > j \wedge S[j] = k \,) \tag{3.7}$$

3.5.3 Properties that Depend on Sender Discipline

Write-Once Channels. A write-once channel is a channel on which the sender sends at most one message. It is characterized by the following property:

$$\textbf{invariant}.(sentn \leq 1) \tag{3.8}$$

From this one property and the property of unordered channels, we can derive the following corollary:

$$delp \wedge del = k \ \textbf{follows} \ sentp \wedge sent = k$$

Monotonic Channels. A monotonic channel is a channel in which the sequence of sent messages is monotonic. We will give here only the properties of monotonically nondecreasing channels. The analogous properties exist for monotonically nonincreasing channels. The fundamental requirement that distinguishes this sender discipline is:

$$\textbf{invariant}.(\forall i \, : \, 0 \leq i < sentn - 1 \, : \, S[i] \leq S[i+1] \,) \tag{3.9}$$

We make use of the function $delm$, the maximum message delivered.

$$delp \Rightarrow delm = (\, \textbf{Max} \, i \, : \, 0 \leq i < deln \, : \, D[i] \,) \tag{3.10}$$

Notice that for monotonic channels with an ordered message-delivery discipline, this function is the same as *del* :

$$delp \Rightarrow delm = del$$

From the fundamental property of monotonic channels and the properties common to all message-delivery disciplines, we can derive the following corollary:

$$delp \wedge delm \geq k \textbf{ follows } sentp \wedge sent \geq k$$

3.6 Simple Certificates for Safety

Simple certificates are restricted forms of the general operators introduced above. They have the advantage of being suitable for automatic translation into run-time checks or warnings. Some of these simple certificates will be given a special syntax (that will be defined using standard Backus-Naur notation [75]).

3.6.1 Monotonicity

We say a variable is monotonic if, in the course of a computation, its value is monotonically increasing or decreasing. This property requires that a partial order be defined on the type of the variable. The ordering relation is typically understood from the type. For example, if the variable is an integer, the ordering relation is the usual $\leq$ operator in the integers. If the variable is a character, the ordering relation might be the alphabetic ordering.

Monotonicity is an important property frequently used in the proofs of distributed (and sequential) systems. A common application of monotonicity is the definition of a metric (also known as a variant function) for a computation. A metric measures the distance, in some sense, to a final goal. In conjunction with boundedness, a monotonic metric is used to establish that a computation never moves further from the final goal.

Monotonicity is a special case of a **next** property. The fact that a variable **x** is monotonically increasing can be expressed as follows:

$$(\forall n :: (x \geq n) \textbf{ next } (x \geq n)) \tag{3.11}$$

We will use the following notation to express the simple certificate of monotonicity:

$$[\textbf{monotonic-up} \mid \textbf{monotonic-down}] .\langle expression\rangle.\langle component\rangle$$

The final argument (the component) will usually be understood from context. For example, Equation 3.11 can be written:

$$\textbf{monotonic-up}.x$$

This notation has the advantage of eliminating the quantification that is required by the **next** formulation.

When the ordering relation on the type of the variable in question is not obvious, it is specified along with the monotonic certificate.

3.6.2 Boundedness

As mentioned above, boundedness is often used in conjunction with monotonicity to help establish termination. The metric must be monotonically increasing (decreasing), but it must also be finite and bounded above (below).[1] As with monotonicity, an ordering relation is required for the type of the bound variable.

Boundedness is a safety property. It is a special case of an **invariant** property. For example, the fact that a variable `x` is bounded above by some value B can be expressed as follows:

$\textbf{invariant}.(x \leq B)$

We do not introduce any special syntax to express boundedness.

When the ordering relation on the type of the variable in question is not obvious, it is specified along with the boundedness certificate.

3.6.3 Unquantified Next

An *unquantified* **next** property is one that does not contain any free variables. For example, the certificate given in Section 3.2.2 to specify that the `Philosopher` component must become thinking after eating was:

$\mathit{eating}\ \textbf{next}\ \mathit{eating} \vee \mathit{thinking}$

This certificate does not require quantification over any variables. The variables `eating` and `thinking` are local variables of `Philosopher` and there are no dummy variables.

No special syntax is required to distinguish these certificates. They are simply a restricted form of general **next** properties.

3.6.4 Functional Next

A *functional* **next** property is one in which the values of the dummy variables are functionally determined by the values of the component variables in the prepredicate of the **next**.

[1] In addition, the metric must not change by less than some positive delta. This condition is frequently obviated by choosing an integer metric.

For example, consider a component with two variables, `x` and `y`, and the following **next** property:

$$(\forall k \; :: \; x \leq y \wedge y = k \;\textbf{next}\; x \leq k \;)$$

The value of the dummy, k, is functionally determined by the value of the component variables (in particular, `y`) in the prepredicate.

Note that this property could be equivalently expressed as:

$$(\forall k \; :: \; x \leq y \wedge y \leq k \;\textbf{next}\; x \leq k \;)$$

This formulation, however, does not functionally determine the value of the dummy for any given values of the component variables.

Consider the generic **next** property p **next** q with dummy variables taken from the set I and component variables taken from the set V. We define such a property to be a *functional* **next** property exactly when:

$$(\forall i \; : \; i \in I \; : \; (\exists f \; :: \; [p \Rightarrow i = f.V] \;) \;)$$

The assymetry in choosing p rather than q to functionally determine the dummy variables is justified by Equation 3.1. From this equation, we know $[p \Rightarrow q]$ and hence if the dummy variables are functionally determined by the postpredicate, they are functionally determined by the prepredicate as well (and hence the property is a functional **next** property in the sense defined above).

We adopt a variation of Hehner's notation [37] for expressing functional **next** properties. We decorate the variables in the prepredicate with an apostrophe placed before the variable name. The **next** can then be replaced by an implication, and the result is a predicate on adjacent pairs of states in a computation. We introduce the operator **adjacent** to distinguish these predicates from predicates on a single state. Let $'p$ denote the predicate that results from decorating the component variables in the predicate p. The property p **next** q can then be written:

$$\textbf{adjacent}.('p \Rightarrow q)$$

Because of the functional determination of the dummy variables, they can be eliminated in this expression by replacing them with a function on the component variables in the prepredicate (*i.e.*, $'p$).

For example, consider the functional **next** at the beginning of this section:

$$(\forall k \; :: \; x \leq y \wedge y = k \;\textbf{next}\; x \leq k \;)$$

Using our decoration of variables, this property can be written:

$$(\forall k :: \textbf{adjacent}.('x \leq 'y \wedge 'y = k \Rightarrow x \leq k))$$

Replacing the dummy with the function on variables in the prepredicate, we have:

$$\textbf{adjacent}.('x \leq 'y \wedge 'y = 'y \Rightarrow x \leq 'y)$$

In this case, the property can be simplified:

$$\textbf{adjacent}.('x \leq 'y \Rightarrow x \leq 'y)$$

Like monotonicity and boundedness, this simple certificate has the advantage of not requiring universal quantification. The use of the keyword **adjacent** and decorated variables is a notational convenience that will be exploited in Chapter 5.

3.7 Simple Certificates for Progress

Recall that a **transient** certificate contains a predicate that is guaranteed not to remain true forever. When used in conjunction with monotonicity and boundedness, transience can establish that a computation eventually reaches a fixed point. If the monotonic bounded metric described above can be shown to eventually change value if it is below the bound, the metric eventually reaches the specified bound.

Like **next** properties, **transient** certificates can contain free variables that are implicitly universally quantified over their range. For example, to express that a metric eventually changes value, the following certificate could be used:

$$\textbf{transient}.(metric = m)$$

where m is a free variable.

For two special cases, however, a free variable and universal quantification are not needed. These two kinds of **transient** properties we will call *unquantified transience* and *functional transience*. Avoiding quantified expressions simplifies the automatic run-time support of these certificates (as will be discussed in Chapter 5).

3.7.1 Unquantified Transience

An unquantified **transient** property is one that does not contain any free variables. For example, the certificate given in Section 3.3.1 to specify that the `Philosopher` component does not eat forever was:

$$\textbf{transient}.eating$$

This certificate does not require quantification over any variables. The variable `eating` is a local variable of `Philosopher` and there are no dummy variables.

No special syntax is required to distinguish these certificates. They are simply a restricted form of general **transient** expressions.

3.7.2 Functional Transience

A frequent use of quantification (especially with regards to metrics) is to express that the value of an expression eventually changes. The free variable is simply used as a place-holder. An example of such a certificate is the example used at the beginning of this section:

$$\textbf{transient}.(metric = m)$$

where m is a free variable.

This certificate is an example of a more general form of **transient** properties in which the free variables are functionally determined by the transient predicate. For any component state, there is at most one value for each free variable such that the transient predicate is true. We call this form of **transient** properties *functional* **transient** properties.

In general, a functional **transient** property with dummy variables taken from the set I and component variables taken from the set V has the form:

$$\textbf{transient}.(\,(\forall\, i \;:\; i \in I \;:\; i = f_i.V\ ) \wedge p.(I, V))$$

We will use the following syntax as an equivalent formulation of the above property:

$$(\forall\, i \;:\; i \in I \;:\; i := f_i.V\ )\ \textbf{in}\ \textbf{transient}.p.(I, V)$$

For example, the certificate concerning the variable `metric` could be written:

$$m := metric\ \textbf{in}\ \textbf{transient}.(metric = m)$$

As another example, consider a metric that is used to establish that a `Philosopher` component does not eat forever. This metric is guaranteed to change, so long as the component remains in the eating state. Such a property is captured by the certificate:

$$m := metric\ \textbf{in}\ \textbf{transient}.(metric = m \wedge eating)$$

Functional **transient** properties have a fundamental similarity to functional **next** properties. Because the free variables are functionally defined by the predicate, universal quantification is not required to express the property.

Chapter 4

Follows Properties

In this chapter, we discuss the certificate **follows** introduced in the previous chapter. This certificate, which to our knowledge is a new property, combines both safety and progress. Some theorems are given that are useful for the manipulation of these properties. As an illustration of the use of **follows**, we succinctly prove a solution to the earliest meeting time problem. Also, as a more involved manipulation of **follows**, we derive the corollaries given as channel properties in the previous chapter (Section 3.5).

4.1 Basic Theorems

In this section, we list some basic theorems for the manipulation of **follows** properties. Most theorems are given without proof, as they can be easily derived from the definition of **follows**. Two theorems (monotonicity and least fixed point) are more involved, so they are proven in the next section.

We begin with the definition (which has already been given in Chapter 3).

Definition.

$$\begin{aligned}(x\ \mathbf{follows}_{\leq}\ y).C \quad \triangleq \quad & (\forall k :: \mathbf{stable}.(x \geq k).C\ ) \wedge (\forall k :: \mathbf{stable}.(y \geq k).C\ ) \\ & \wedge \mathbf{invariant}.(x \leq y).C \wedge (\forall k :: (y \geq k \leadsto x \geq k).C\ ) \\ & \wedge (\forall k :: (y \leq k \wedge x \leq k\ \mathbf{next}\ x \leq k).C\ )\end{aligned}$$

The subscript on the **follows** operator will be omitted when the ordering relation is clear from context. For the following properties, the ordering relation will be understood to be $\leq$. Also, we will use x, y, and z for program variables and j and k for free variables.

4.1.1 Properties for Posets

A poset is a set of elements with an ordering relation that is reflexive, transitive, and antisymmetric. An example of a poset is the set of sequences ordered by the prefix ordering relation.

When the ordering relation of **follows** (*e.g.*, $\leq$) defines a poset on the types of the variables, it enjoys the following properties.

Constants. For free variables j and k (x is a program variable), we have:

$$k \textbf{ follows } j \quad \equiv \quad \textbf{invariant}.(k = j) \tag{4.1}$$

$$k \textbf{ follows } x \quad \equiv \quad \textbf{invariant}.(x = k) \tag{4.2}$$

$$x \textbf{ follows } k \quad \equiv \quad \textbf{invariant}.(x = k) \tag{4.3}$$

Transitivity.

$$(x \textbf{ follows } y) \wedge (y \textbf{ follows } z) \quad \Rightarrow \quad (x \textbf{ follows } z) \tag{4.4}$$

Reflexivity.

$$(x \textbf{ follows } x) \quad \equiv \quad (\exists\, k :: \textbf{invariant}.(x = k)\ ) \tag{4.5}$$

Antisymmetry.

$$(x \textbf{ follows } y) \wedge (y \textbf{ follows } x) \quad \Rightarrow \quad \textbf{invariant}.(x = y) \tag{4.6}$$

Monotonicity.

$$(f \textit{ is monotonic}) \wedge (x \textbf{ follows } y) \quad \Rightarrow \quad (f.x \textbf{ follows } f.y) \tag{4.7}$$

Stable Fixed Point. An element k is said to be a *fixed point* of a function f when $k = f.k$. As a shorthand, we define the set FP of fixed points, $FP = \{\, k \ : \ k = f.k \ : \ k \,\}$

$$(x \textbf{ follows } f.x) \quad \Rightarrow \quad (\forall\, k \ : \ k \in FP \ : \ \textbf{stable}.(x = k)\ ) \tag{4.8}$$

Least Fixed Point. As a shorthand, we define the set $FP.x_0$ of fixed points above an element x_0 :

$$FP.x_0 = \{\, k \ : \ k \geq x_0 \wedge k = f.k \ : \ k \,\}$$

An element k is said to be in the *finite closure* of an element x_0 and a function f when k can be otained by a finite number of applications of f (*i.e.*, f^i, for some finite i) to x_0. As a shorthand, we define the set $FC.x_0$ of elements that are in this finite closure:

$$FC.x_0 = \{\, k \;:\; (\exists\, i \;:\; i \geq 0 \;:\; k = f^i.x0 \;) \;:\; k \;\}$$

The following theorem gives a useful progress property as a consequence of a **follows** property.

$$\begin{aligned} &(f \ \textit{is monotonic}) \wedge ((FP \cap FC).x_0 \neq \phi) \wedge (x \ \textbf{follows}\ f.x) \\ &\quad \Rightarrow \quad (x = x_0 \leadsto x = (\,\textbf{Min}\, k \;:\; k \in FP.x_0 \;:\; k\;)) \end{aligned} \tag{4.9}$$

Informally, this theorem expresses a kind of induction on the poset. It is interesting to note that well foundedness of the poset is neither necessary nor sufficient for the application of the theorem. There is another characterization of posets, however, that is easy to verify (independent of a function f) and that arises frequently in practice. This characterization is given next.

A *chain* is a sequence of elements that can be arranged such that each element is below the next element. In a poset where every chain between any two comparable elements is finite, there exists a fixed point above x_0 only when there exists a fixed point above x_0 that is the result of the application of some finite closure to x_0. Thus, in any such poset, the existence of a fixed point above x_0 is sufficient for the application of the theorem. The set of integers is an example of such a poset. For such a poset, the least fixed point theorem can be restated more simply as follows:

$$\begin{aligned} &(f \ \textit{is monotonic}) \wedge (FP.x_0 \neq \phi) \wedge (x \ \textbf{follows}\ f.x) \\ &\quad \Rightarrow \quad (x = x_0 \leadsto x = (\,\textbf{Min}\, k \;:\; k \in FP.x_0 \;:\; k\;)) \end{aligned} \tag{4.10}$$

4.1.2 Properties for Lattices

A poset is called a *lattice* if every finite nonempty set has a greatest lower bound (or "meet," denoted $\downarrow$) and a least upper bound (or "join," denoted $\uparrow$) [7, 97]. Equivalently, a poset is a lattice exactly when:

$$(\forall\, x, y \;::\; (\exists\, z \;::\; z = x \downarrow y\;) \wedge (\exists\, z \;::\; z = x \uparrow y\;)\;)$$

Sequences ordered by prefix ordering do not form a lattice because a least upper bound is not defined for all pairs of sequences. (In particular, two sequences in which neither is a prefix of the other have no common upper bound.) An example of a lattice is the set of integers ordered by the "at most" relation.

For posets that are lattices, **follows** enjoys the following properties, in addition to those given in Section 4.1.1.

Strengthening / Weakening.

$$\begin{aligned}&(\forall k :: \textbf{stable}.(z \geq k)) \wedge (x \ \textbf{follows} \ y)\\ &\quad \Rightarrow \ (x \downarrow z \ \textbf{follows} \ y \downarrow z) \wedge (x \uparrow z \ \textbf{follows} \ y \uparrow z)\end{aligned} \tag{4.11}$$

Junctivity (Finite). For finite nonempty I,

$$\begin{aligned}&(\forall i : i \in I : x_i \ \textbf{follows} \ y_i)\\ &\quad \Rightarrow \quad (\downarrow \{ i : i \in I : x_i \} \ \textbf{follows} \downarrow \{ i : i \in I : y_i \})\\ &\qquad \wedge (\uparrow \{ i : i \in I : x_i \} \ \textbf{follows} \uparrow \{ i : i \in I : y_i \})\end{aligned} \tag{4.12}$$

Union (Finite). For finite nonempty I,

$$\begin{aligned}&(\forall i : i \in I : x_i \ \textbf{follows} \ y)\\ &\quad \Rightarrow \ (\downarrow \{ i : i \in I : x_i \} \ \textbf{follows} \uparrow \{ i : i \in I : x_i \})\end{aligned} \tag{4.13}$$

Intersection (Finite). For finite nonempty I,

$$\begin{aligned}&(\forall i : i \in I : x \ \textbf{follows} \ y_i)\\ &\quad \Rightarrow \ (\downarrow \{ i : i \in I : y_i \} \ \textbf{follows} \uparrow \{ i : i \in I : y_i \})\end{aligned} \tag{4.14}$$

4.1.3 Properties for Complete Lattices

A lattice is said to be *complete* if all sets (including infinite and empty ones) have a least upper bound and a greatest lower bound. Complete lattices always have a *top* (an element above all others) and a *bottom* (an element below all others).

The set of integers ordered by “at most” does not form a complete lattice because there is no top or bottom. (The empty set does not have a least upper bound or a greatest lower bound.) An example of a complete lattice is the powerset of a finite set S, ordered by subset inclusion. Any two elements have a least upper bound (the union) and a greatest lower bound (the intersection). The top is the set S and the bottom is the empty set.

For complete lattices, the finite properties given above are universal.

Junctivity (Universal).

$$(\forall\, i\ ::\ x_i\ \textbf{follows}\ y_i\ )\quad\Rightarrow\quad \begin{array}{l}(\downarrow\{\, i\ ::\ x_i\ \}\,\textbf{follows}\downarrow\{\, i\ ::\ y_i\ \})\\ \wedge\,(\uparrow\{\, i\ ::\ x_i\ \}\,\textbf{follows}\uparrow\{\, i\ ::\ y_i\ \})\end{array} \tag{4.15}$$

Union (Universal).

$$(\forall\, i\ ::\ x_i\ \textbf{follows}\ y\ )\quad\Rightarrow\quad(\downarrow\{\, i\ ::\ x_i\ \}\,\textbf{follows}\uparrow\{\, i\ ::\ x_i\ \}) \tag{4.16}$$

Intersection (Universal).

$$(\forall\, i\ ::\ x\ \textbf{follows}\ y_i\ )\quad\Rightarrow\quad(\downarrow\{\, i\ ::\ y_i\ \}\,\textbf{follows}\uparrow\{\, i\ ::\ y_i\ \}) \tag{4.17}$$

4.2 Proofs of Basic Theorems

The theorems listed above can all be derived from the definition of **follows**. The complete proofs for two of these theorems (monotonicity and least fixed point) are given here as an illustration of how such a proof can be carried out. These two theorems arise frequently in proofs involving **follows**.

4.2.1 Monotonicity

$$(f\ \textit{is monotonic})\wedge(x\ \textbf{follows}\ y)\quad\Rightarrow\quad(f.x\ \textbf{follows}\ f.y)$$

We prove the consequent, $f.x$ **follows** $f.y$, by proving each of the conjuncts in the definition of **follows**.

$$\begin{array}{rcl} f.x\ \textbf{follows}\ f.y & \equiv & (\forall\, k\ ::\ \textbf{stable}.(f.x\geq k)\ )\wedge(\forall\, k\ ::\ \textbf{stable}.(f.y\geq k)\ )\\ & & \wedge\ \textbf{invariant}.(f.x\leq f.y)\wedge(\forall\, k\ ::\ f.y\geq k\leadsto f.x\geq k\ )\\ & & \wedge\ (\forall\, k\ ::\ f.y\leq k\wedge f.x\leq k\ \textbf{next}\ f.x\leq k\ )\end{array}$$

Prove:

$$(\forall\, k\ ::\ \textbf{stable}.(f.x\geq k)\ )$$

Proof:

$$\begin{array}{ll}
& f.x = k \\
\equiv & \quad \{ \text{ choice of j } \} \\
& x = j \;\wedge\; f.j = k \\
\textbf{next} & \{ \text{ assumption : } \textbf{stable}.(x \geq k) \quad \} \\
& x \geq j \;\wedge\; f.j = k \\
\Rightarrow & \quad \{ \; f \text{ is monotonic } \} \\
& f.x \geq f.j \;\wedge\; f.j = k \\
\Rightarrow & \quad \{ \text{ calculus } \} \\
& f.x \geq k
\end{array}$$

□

Prove:

$$(\forall k \;::\; \textbf{stable}.(f.y \geq k)\;)$$

Proof: Same as above.

□

Prove:

$$\textbf{invariant}.(f.x \leq f.y)$$

Proof:

$$\begin{array}{ll}
& \mathit{true} \\
\equiv & \quad \{ \text{ assumption: } \textbf{invariant}.(x \leq y) \quad \} \\
& \textbf{invariant}.(x \leq y) \\
\Rightarrow & \quad \{ \; f \text{ is monotonic } \} \\
& \textbf{invariant}.(f.x \leq f.y)
\end{array}$$

□

Prove:

$$f.y \geq k \leadsto f.x \geq k$$

Proof: There exists a j such that

$$
\begin{array}{ll}
 & f.y \geq k \\
\equiv & \quad \{ \text{ choice of } j \ \} \\
 & y = j \ \wedge \ f.y \geq k \\
\equiv & \quad \{ \text{ calculus } \} \\
 & y = j \ \wedge \ f.y \geq k \ \wedge \ f.j \geq k \\
\Rightarrow & \quad \{ \text{ calculus } \} \\
 & y \geq j \ \wedge \ f.y \geq k \ \wedge \ f.j \geq k \\
\leadsto & \quad \{ \text{ assumption : } \ y \geq k \ \leadsto \ x \geq k \text{ , with } \ k := j \ \} \\
 & x \geq j \ \wedge \ f.j \geq k \\
\Rightarrow & \quad \{ \ f \text{ is monotonic } \} \\
 & f.x \geq f.j \ \wedge \ f.j \geq k \\
\Rightarrow & \quad \{ \text{ calculus } \} \\
 & f.x \geq k
\end{array}
$$

□

Prove:

$$f.y \leq k \wedge f.x \leq k \ \mathbf{next} \ f.x \leq k$$

Proof: There exists a j such that

$$
\begin{array}{ll}
 & f.y \leq k \ \wedge \ f.x \leq k \\
\Rightarrow & \quad \{ \text{ calculus } \} \\
 & f.y \leq k \\
\equiv & \quad \{ \text{ choice of j } \} \\
 & y = j \ \wedge \ f.y \leq k \\
\equiv & \quad \{ \text{ calculus } \} \\
 & y = j \ \wedge \ f.j \leq k \\
\equiv & \quad \{ \text{ assumption: } \mathbf{invariant}.(x \leq y) \ \} \\
 & y = j \ \wedge \ f.j \leq k \ \wedge \ x \leq y \\
\equiv & \quad \{ \text{ calculus } \} \\
 & y = j \ \wedge \ f.j \leq k \ \wedge \ x \leq j \\
\mathbf{next} & \quad \{ \text{ assumption: } \ y \leq k \ \wedge \ x \leq k \ \mathbf{next} \ \ x \leq k \ \} \\
 & f.j \leq k \ \wedge \ x \leq j
\end{array}
$$

$$\begin{array}{ll} \Rightarrow & \{ \text{ f is monotonic } \} \\ & f.j \leq k \ \wedge f.x \leq f.j \\ \Rightarrow & \{ \text{ calculus } \} \\ & f.x \leq k \end{array}$$

□

4.2.2 Least Fixed Point

$$(f \ \textit{is monotonic}) \wedge ((FP \cap FC).x_0 \neq \phi) \wedge (x \ \textbf{follows} \ f.x)$$
$$\Rightarrow \quad (x = x_0 \leadsto x = (\, \textbf{Min}\, k \ : \ k \in FP.x_0 \ : \ k \,))$$

We define a constant m as follows:

$$m = (\, \textbf{Min}\, k \ : \ k \in (FP \cap FC).x_0 \ : \ k \,)$$

After justifying the use of **min** in the definition of m, we will prove the following results:

$$m = (\, \textbf{Min}\, k \ : \ k \in FP.x_0 \ : \ k \,) \tag{4.18}$$
$$x = x_0 \leadsto x \geq m \tag{4.19}$$
$$\textbf{stable}.(x \leq m) \tag{4.20}$$

It is clear that the least fixed point theorem follows from the conjunction of these results.

Prove: $FC.x_0$ is a well-founded set (and hence any nonempty subset has a minimum element, as in the definition of m).

Proof: By induction, we show $(\forall i \ : \ i \geq 0 \ : \ f^i.x_0 \leq f^{i+1}.x_0 \,)$.

Base case ($i = 0$).

$$\begin{array}{ll} & \textit{true} \\ \equiv & \{ \ \textbf{invariant}.(x \leq f.x) \ \} \\ & \textbf{invariant}.(x \leq f.x) \\ \Rightarrow & \{ \text{ definition of } \textbf{invariant} \ \} \\ & \textbf{initially}.(x \leq f.x) \\ \equiv & \{ \text{ assumption: } \textbf{initially}.(x = x_0) \ \} \\ & \textbf{initially}.(x \leq f.x) \ \wedge \ \textbf{initially}.(x = x_0) \end{array}$$

$$
\begin{array}{ll}
\Rightarrow & \{ \text{ calculus } \} \\
& \textbf{initially}.(x_0 \leq f.x_0) \\
\equiv & \{ \ x_0 \text{ is a constant } \} \\
& x_0 \leq f.x_0 \\
\equiv & \{ \text{ definition of } f^0 \ \} \\
& f^0.x_0 \leq f.x_0
\end{array}
$$

Inductive step.

$$
\begin{array}{ll}
& f^i.x_0 \leq f^{i+1}.x_0 \\
\Rightarrow & \{ \ f \text{ is monotonic } \} \\
& f^{i+1}.x_0 \leq f^{i+2}.x_0
\end{array}
$$

Now $(FP \cap FC).x_0$ is a nonempty (from the assumptions) subset of a well-founded set. Hence, it has a minimum (*i.e.*, m is well defined).

□

Prove (4.18):

$$m = (\,\mathbf{Min}\, k \ : \ k \in FP.x_0 \ : \ k\ )$$

Proof: First we show by induction:

$$(\forall j \ : \ j \in FC.x_0 \ : \ (\forall k \ : \ k \in FP.x_0 \ : \ k \geq j \ ) \ )$$

Base case ($i = 0$). $(\forall k \ : \ k \in FP.x_0 \ : \ k \geq f^0.x_0 \ )$

$$
\begin{array}{ll}
& (\forall k \ : \ k \in FP.x_0 \ : \ k \geq f^0.x_0 \ ) \\
\equiv & \{ \text{ definition of } f^0 \ \} \\
& (\forall k \ : \ k \in FP.x_0 \ : \ k \geq x_0 \ ) \\
\equiv & \{ \text{ definition of } FP.x_0 \ \} \\
& \mathit{true}
\end{array}
$$

Inductive step.

$$
\begin{array}{ll}
& (\forall k \ : \ k \in FP.x_0 \ : \ k \geq f^i.x_0 \ ) \\
\Rightarrow & \{ \ f \text{ is monotonic } \} \\
& (\forall k \ : \ k \in FP.x_0 \ : \ f.k \geq f^{i+1}.x_0) \ ) \\
\Rightarrow & \{ \text{ definition of } FP.x_0 \ \} \\
& (\forall k \ : \ k \in FP.x_0 \ : \ k \geq f^{i+1}.x_0 \ )
\end{array}
$$

In particular for m (since $m \in FC.x_0$), we have:

$$(\forall k \; : \; k \in FP.x_0 \; : \; k \geq m \;)$$

But $m \in FP.x_0$ (by definition of m), so:

$$m = (\mathbf{Min}\, k \; : \; k \in FP.x_0 \; : \; k \;)$$

□

Prove (4.19):

$$x \geq x_0 \leadsto x \geq m$$

Proof: First we show:

$$(\forall i \; : \; i \geq 0 \; : \; x \geq f^{i}.x_0 \leadsto x \geq f^{i+1}.x_0 \;)$$

Consider any $i \geq 0$,

$$\begin{array}{ll} & x \geq f^{i}.x_0 \\ \Rightarrow & \quad \{ \; f \text{ is monotonic } \} \\ & f.x \geq f^{i+1}.x_0 \\ \leadsto & \quad \{ \text{ assumption : } f.x \geq k \leadsto x \geq k \; \} \\ & x \geq f^{i+1}.x_0 \end{array}$$

Inducting on integers, this gives the property:

$$(\forall k \; : \; k \in FC.x_0 \; : \; x \geq x_0 \leadsto x \geq k \;)$$

And hence (since m is in $FC.x_0$):

$$x \geq x_0 \leadsto x \geq m$$

□

Prove (4.20):

$$\mathbf{stable}.(x \leq m)$$

Proof:

$$
\begin{array}{ll}
 & x \le m \\
\equiv & \quad \{ \ f \ \text{is monotonic} \ \} \\
 & x \le m \ \wedge \ f.x \le f.m \\
\equiv & \quad \{ \ \text{definition of} \ m \ : \ m \in FP.x_0 \ \} \\
 & x \le m \ \wedge \ f.x \le m \\
\textbf{next} & \quad \{ \ \text{assumption:} \ f.x \le k \ \wedge \ x \le k \ \textbf{next} \ x \le k \ \} \\
 & x \le m
\end{array}
$$

□

4.3 The Earliest Meeting Time

4.3.1 Problem Definition

The earliest meeting time problem has been discussed in [19]. The problem is to schedule a meeting for a group of people at the earliest time that is acceptable to every member of the group. Time is considered to be a nonnegative integer value. Associated with every member i is a function f_i that maps any time t to the first available time for member i at or after time t. From the definition of f_i, it follows that f_i is monotonic and moreover that $(\forall t :: t \le f_i.t)$. An example of these functions for a group of three people is plotted in Figure 4.1.

Initially, the program variable t is 0. The progress requirement is that eventually this variable is set to the earliest meeting time, provided one exists. The safety requirement is that this be a fixed point of the system. More formally, the problem specification is, assuming $(\exists k :: (\forall i :: k = f_i.k))$, to establish:

$$t = 0 \leadsto t = (\mathbf{Min}\, k : (\forall i :: k = f_i.k) : k) \tag{4.21}$$

$$\mathbf{stable}.(t = (\mathbf{Min}\, k : (\forall i :: k = f_i.k) : k)) \tag{4.22}$$

4.3.2 A Solution

We will consider here a specification of a solution. The solution can be implemented by a distributed system in our model of computation by a central component that keeps track of the current proposed time, and a collection of components, one for each member of the group.

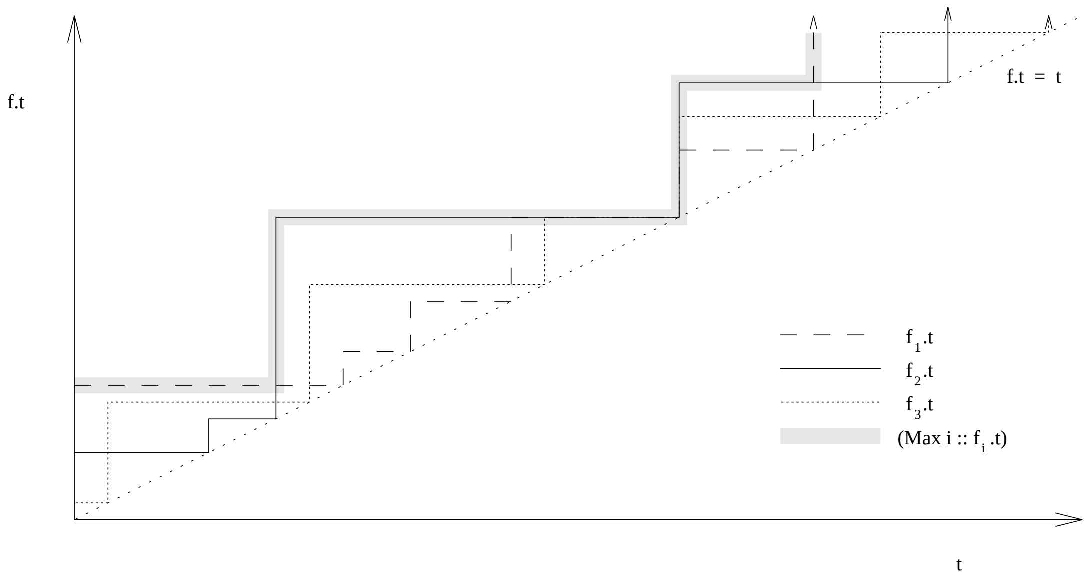

Figure 4.1: Some example earliest available meeting time functions for a group of people.

The specification of our solution is:

$$\textbf{initially}.(t = 0) \tag{4.23}$$

$$t \ \textbf{follows} \ (\textbf{Max}\, i \ :: \ t_i \ ) \tag{4.24}$$

$$(\forall\, i \ :: \ t_i \ \textbf{follows} \ f_i.t \ ) \tag{4.25}$$

Operationally, each component that corresponds to a member of the group is responsible for updating t_i with the next available time for component i at or after time t. The central component takes the maximum of these values and updates t.

4.3.3 Proof of Solution

Lemma 4.1

$$t \ \textbf{follows} \ (\textbf{Max}\, i \ :: \ f_i.t \ )$$

Proof:

$$\begin{array}{ll} & \mathit{true} \\ \equiv & \quad \{ \ \text{property} : \ (\forall\, i \ :: \ t_i \ \textbf{follows} \ f_i.t \ ) \ \} \\ & (\forall\, i \ :: \ t_i \ \textbf{follows} \ f_i.t \ ) \\ \Rightarrow & \quad \{ \ \text{finite junctivity of } \textbf{follows} \ \} \end{array}$$

$$
\begin{array}{ll}
& (\,\mathbf{Max}\, i :: t_i\,)\ \ \mathbf{follows}\ \ (\,\mathbf{Max}\, i :: f_i.t\,) \\
\equiv & \quad \{\ \text{property} : t\ \ \mathbf{follows}\ \ (\,\mathbf{Max}\, i :: t_i\,), \text{ and transitivity of } \mathbf{follows}\ \} \\
& t\ \ \mathbf{follows}\ \ (\,\mathbf{Max}\, i :: f_i.t\,)
\end{array}
$$

□

We now use the stable fixed point theorem and the least fixed point theorem (since all the f_i are monotonic and a fixed point exists by assumption) to conclude:

$$
\begin{array}{l}
t = 0 \leadsto t = (\,\mathbf{Min}\, k : k = (\,\mathbf{Max}\, i :: f_i.k\,) : k\,) \\
\wedge\, \mathbf{stable}.(t = (\,\mathbf{Min}\, k : k = (\,\mathbf{Max}\, i :: f_i.k\,) : k\,))
\end{array}
$$

Now this result is equivalent to the required specification, as established from the following equivalence:

$$
\begin{array}{ll}
& k = (\,\mathbf{Max}\, i :: f_i.k\,) \equiv (\,\forall\, i :: k = f_i.k\,) \\
\equiv & \quad \{\ \text{definition of } \ \mathbf{max}\ \} \\
& (\,\forall\, i :: k \geq f_i.k\,) \wedge (\,\exists\, i :: k = f_i.k\,) \equiv (\,\forall\, i :: k = f_i.k\,) \\
\equiv & \quad \{\ \text{property of } f_i : k \leq f_i.k\ \} \\
& (\,\forall\, i :: k = f_i.k\,) \wedge (\,\exists\, i :: k = f_i.k\,) \equiv (\,\forall\, i :: k = f_i.k\,) \\
\equiv & \quad \{\ \text{calculus}\ \} \\
& (\,\forall\, i :: k = f_i.k\,) \Rightarrow (\,\exists\, i :: k = f_i.k\,) \\
\equiv & \quad \{\ \text{calculus, for a nonemtpy group}\ \} \\
& \mathit{true}
\end{array}
$$

□

Notice that the entire specification (both safety and progress) is established from a single **follows** property (as given in Lemma 4.1).

4.4 Proofs of Channel Properties

In Chapter 3, the properties that define channel behavior were given, along with some corollaries derivable from these properties. In this section, we show the derivation as an exercise in the use of **follows** properties. We begin with a reminder of the basic definitions common to all channel types before proving the corollaries for unordered channels, ordered channels, write-once channels, and monotonic channels.

4.4.1 Definitions Common to All Channel Types

These definitions have already been given in Section 3.5.1. They are repeated here for convenience.

$$deln \;=\; |D| \tag{4.26}$$

$$sentn \;=\; |S| \tag{4.27}$$

$$delp \;\equiv\; deln > 0 \tag{4.28}$$

$$sentp \;\equiv\; sentn > 0 \tag{4.29}$$

$$delp \;\Rightarrow\; del = D[deln - 1] \tag{4.30}$$

$$sentp \;\Rightarrow\; sent = S[sentn - 1] \tag{4.31}$$

4.4.2 Properties that Depend on Message-Delivery Discipline

Unordered Channels.

Property:

$$D \ \mathbf{follows}_{\sqsubseteq} \ S \tag{4.32}$$

Lemma 4.2 *Length ($||$) of sequences is monotonic with respect to the multiset subset ($\sqsubseteq$) ordering.*

$$X \sqsubseteq Y \;\Rightarrow\; (|X| \leq |Y|)$$

Proof: The lemma follows directly from the definition of the multiset subset ($\sqsubseteq$) ordering on sequences.

□

Corollary:

$$deln \ \mathbf{follows} \ sentn \tag{4.33}$$

Proof:

$$\begin{array}{ll} & D \ \mathbf{follows}_{\sqsubseteq} \ S \\ \Rightarrow & \quad \{ \text{ Lemma 4.2 : length is monotonic with respect to } \sqsubseteq \ \} \\ & |D| \ \mathbf{follows} \ |S| \end{array}$$

$$
\begin{array}{ll}
\equiv & \{\ 4.26:\ deln = |D|\text{, and } 4.27:\ sentn = |S|\ \} \\
& deln\ \textbf{follows}\ sentn
\end{array}
$$

□

Corollary:

$$delp\ \textbf{follows}\ sentp \tag{4.34}$$

Proof:

$$
\begin{array}{ll}
& deln\ \textbf{follows}\ sentn \\
\Rightarrow & \{\ \text{monotonicity of predicate } x \geq 1 \text{ with respect to } \leq\ \} \\
& deln \geq 1\ \textbf{follows}\ sentn \geq 1 \\
\equiv & \{\ 4.28:\ delp\ \equiv\ deln > 0\text{, and } 4.29:\ sentp\ \equiv\ sentn > 0\ \} \\
& delp\ \textbf{follows}\ sentp
\end{array}
$$

□

Ordered Channels.

Property:

$$D\ \textbf{follows}_{\preceq}\ S \tag{4.35}$$

Lemma 4.3 *The multiset subset ($\sqsubseteq$) ordering on sequences is monotonic with respect to the prefix ($\preceq$) ordering.*

$$X \preceq Y \Rightarrow X \sqsubseteq Y$$

Proof:

$$
\begin{array}{ll}
& (\#\, i\ :\ 0 \leq i < |X|\ :\ X[i] = k\ ) \\
= & \{\ \text{definition of } \preceq\ :\ X \preceq Y\ \Rightarrow\ (\forall\, i\ :\ 0 \leq i < |X|\ :\ X[i] = Y[i]\ )\ \} \\
& (\#\, i\ :\ 0 \leq i < |X|\ :\ Y[i] = k\ ) \\
\leq & \{\ \text{definition of } \preceq\ :\ X \preceq Y\ \Rightarrow\ |X| \leq |Y|\ \} \\
& (\#\, i\ :\ 0 \leq i < |Y|\ :\ Y[i] = k\ )
\end{array}
$$

□

Corollary:

$$D \ \mathbf{follows}_{\sqsubseteq} \ S$$

Proof:

$$\begin{array}{ll} & D \ \mathbf{follows}_{\preceq} \ S \\ \Rightarrow & \quad \{ \text{ Lemma 4.3 : monotonicity of } \sqsubseteq \text{ with respect to } \preceq \ \} \\ & D \ \mathbf{follows}_{\sqsubseteq} \ S \end{array}$$

□

Lemma 4.4 *For all* k *and all* $j \geq 0$ *, the predicate* $|X| > j \ \wedge \ X[j] = k$ *on a sequence* X *is monotonic with respect to the prefix (* $\preceq$ *) ordering on sequences.*

$$X \preceq Y \Rightarrow (|X| > j \wedge X[j] = k \Rightarrow |Y| > j \wedge Y[j] = k)$$

Proof: The lemma follows directly from the definition of the prefix ($\preceq$) ordering on sequences.

□

Corollary:

$$(\forall j \ : \ j \geq 0 \ : \ deln > j \wedge D[j] = k \ \mathbf{follows} \ sentn > j \wedge S[j] = k \ )$$

Proof:

$$\begin{array}{ll} & D \ \mathbf{follows}_{\preceq} \ S \\ \Rightarrow & \quad \{ \text{ Lemma 4.4 : monotonicity of predicate with respect to } \preceq \ \} \\ & (\forall j \ : \ j \geq 0 \ : \ deln > j \ \wedge \ D[j] = k \ \ \mathbf{follows} \ \ sentn > j \ \wedge \ S[j] = k \ ) \end{array}$$

□

4.4.3 Properties that Depend on Sender Discipline

Write-Once Channels.

Property:

$$\mathbf{invariant}.(sentn \leq 1) \tag{4.36}$$

Lemma 4.5

$$\mathbf{invariant}.(deln \leq 1)$$

Proof:

$$\begin{array}{ll} & true \\ \equiv & \quad \{ \ 4.36 : \ \mathbf{invariant}.(sentn \le 1) \ \} \\ & \mathbf{invariant}.(sentn \le 1) \\ \Rightarrow & \quad \{ \ 4.32 : \ deln \ \mathbf{follows} \ sentn \ \} \\ & \mathbf{invariant}.(deln \le 1) \end{array}$$

□

Lemma 4.6

$$delp \equiv deln = 1$$

$$sentp \equiv sentn = 1$$

Proof: The proof follows immediately from Lemma 4.5, the property 4.36 of write-once channels, and the common channel definitions.

□

Lemma 4.7 *For all k, the predicate $|X| = 1 \wedge X[0] = k$ on a sequence X is monotonic with respect to the multiset subset ($\sqsubseteq$) ordering on sequences, for sequences of length less than or equal to 1.*

$$|X| \le 1 \wedge |Y| \le 1 \wedge X \sqsubseteq Y \Rightarrow (|X| = 1 \wedge X[0] = k \Rightarrow |Y| = 1 \wedge Y[0] = k)$$

Proof:

$$\begin{array}{ll} & |X| = 1 \ \wedge \ X[0] = k \\ \Rightarrow & \quad \{ \ \text{assumption} : \ X \sqsubseteq Y \ \} \\ & |Y| \ge 1 \ \wedge \ (\exists j \ : \ |Y| > j \ge 0 \ : \ Y[j] = k \ ) \\ \equiv & \quad \{ \ \text{assumption} : \ |Y| \le 1 \ \} \\ & |Y| = 1 \ \wedge \ (\exists j \ : \ 1 > j \ge 0 \ : \ Y[j] = k \ ) \\ \equiv & \quad \{ \ \text{calculus} \ \} \\ & |Y| = 1 \ \wedge \ Y[0] = k \end{array}$$

□

Corollary:

$$delp \wedge del = k \ \mathbf{follows} \ sentp \wedge sent = k$$

Proof:

$$
\begin{array}{ll}
 & D \ \mathbf{follows}_{\sqsubseteq} \ S \\
\equiv & \quad \{ \ 4.36 : \ \mathbf{invariant}.(sentn \le 1) \ \} \\
 & (D \ \mathbf{follows}_{\sqsubseteq} \ S) \wedge \mathbf{invariant}.(sentn \le 1) \\
\equiv & \quad \{ \ \text{Lemma 4.5} : \ \mathbf{invariant}.(deln \le 1) \ \} \\
 & (D \ \mathbf{follows}_{\sqsubseteq} \ S) \wedge \mathbf{invariant}.(sentn \le 1 \ \wedge \ deln \le 1) \\
\equiv & \quad \{ \ 4.26 : \ deln = |D| \text{, and } 4.27 : \ sentn = |S| \ \} \\
 & (D \ \mathbf{follows}_{\sqsubseteq} \ S) \wedge \mathbf{invariant}.(|S| \le 1 \ \wedge \ |D| \le 1) \\
\Rightarrow & \quad \{ \ \text{Lemma 4.7 : predicate is monotonic on sequences of length less than} \\
 & \quad\quad \text{or equal to 1} \ \} \\
 & |D| = 1 \ \wedge \ D[0] = k \ \ \mathbf{follows} \ \ |S| \ \wedge \ S[0] = k \\
\equiv & \quad \{ \ 4.26 : \ deln \ = \ |D| \text{, and } 4.27 : \ sentn \ = \ |S| \ \} \\
 & deln = 1 \ \wedge \ D[0] = k \ \ \mathbf{follows} \ \ sentn = 1 \ \wedge \ S[0] = k \\
\equiv & \quad \{ \ \text{calculus} \ \} \\
 & deln = 1 \ \wedge \ D[deln - 1] = k \ \ \mathbf{follows} \ \ sentn = 1 \ \wedge \ S[sentn - 1] = k \\
\equiv & \quad \{ \ \text{Lemma 4.6} : \ delp = (deln = 1) \text{, and } \ sentp = (sentn = 1) \ \} \\
 & delp \ \wedge \ D[deln - 1] = k \ \ \mathbf{follows} \ \ sentp \ \wedge \ S[sentn - 1] = k \\
\equiv & \quad \{ \ 4.30 : \ delp \ \Rightarrow \ del = D[deln - 1] \ \} \\
 & delp \ \wedge \ del = k \ \ \mathbf{follows} \ \ sentp \ \wedge \ S[sentn - 1] = k \\
\equiv & \quad \{ \ 4.31 : \ sentp \ \Rightarrow \ sent = S[sentn - 1] \ \} \\
 & delp \ \wedge \ del = k \ \ \mathbf{follows} \ \ sentp \ \wedge \ sent = k
\end{array}
$$

□

Monotonic Channels.

Property:

$$\mathbf{invariant}.(\forall i \ : \ 0 \le i < sentn - 1 \ : \ S[i] \le S[i+1] \ ) \tag{4.37}$$

Definition:

$$delp \Rightarrow delm = (\ \mathbf{Max}\, i \ : \ 0 \le i < deln \ : \ D[i] \ ) \tag{4.38}$$

Lemma 4.8

$$
\begin{array}{l}
(\forall k \ : \ 0 < k \le |X| \ : \ (\forall i \ : \ 0 \le i < k - 1 \ : \ X[i] \le X[i+1] \ ) \\
\qquad\qquad\qquad \Rightarrow X[k-1] = (\ \mathbf{Max}\, i \ : \ 0 \le i < k \ : \ X[i] \ ) \ )
\end{array}
$$

Proof:

$$
\begin{array}{ll}
& (\forall i \ : \ 0 \leq i < k-1 \ : \ X[i] \leq X[i+1] \ ) \\
\Rightarrow & \quad \{ \text{ calculus : transitivity of } \leq \text{ , and } k > 0 \ \} \\
& (\forall i \ : \ 0 \leq i < k-1 \ : \ X[i] \leq X[k-1] \ ) \\
\equiv & \quad \{ \text{ calculus : reflexivity of } \leq \text{ , and } k > 0 \ \} \\
& (\forall i \ : \ 0 \leq i < k \ : \ X[i] \leq X[k-1] \ ) \\
\equiv & \quad \{ \text{ calculus : definition of } \mathbf{max} \text{ , and } k > 0 \ \} \\
& X[k-1] = (\, \mathbf{Max}\, i \ : \ 0 \leq i < k \ : \ X[i] \ )
\end{array}
$$

□

Lemma 4.9

$$sentp \Rightarrow sent = (\, \mathbf{Max}\, i \ : \ 0 \leq i < sentn \ : \ S[i] \ )$$

Proof:

$$
\begin{array}{ll}
& sentp \\
\equiv & \quad \{ \text{ 4.29: } sentp \ \equiv \ sentn > 0 \ \} \\
& sentn > 0 \\
\equiv & \quad \{ \text{ 4.37 : } \ (\forall i \ : \ 0 \leq i < sentn - 1 \ : \ S[i] \leq S[i+1] \ ) \ \} \\
& sentn > 0 \ \wedge \ (\forall i \ : \ 0 \leq i < sentn - 1 \ : \ S[i] \leq S[i+1] \ ) \\
\Rightarrow & \quad \{ \text{ Lemma 4.8 with k := sentn } \} \\
& sentn > 0 \ \wedge \ S[sentn-1] = (\, \mathbf{Max}\, i \ : \ 0 \leq i < sentn \ : \ S[i] \ ) \\
\equiv & \quad \{ \text{ 4.29: } sentp \ \equiv \ sentn > 0 \ \} \\
& sentp \ \wedge \ S[sentn-1] = (\, \mathbf{Max}\, i \ : \ 0 \leq i < sentn \ : \ S[i] \ ) \\
\equiv & \quad \{ \text{ 4.31: } sentp \ \Rightarrow \ sent = S[sentn-1] \ \} \\
& sentp \ \wedge \ sent = (\, \mathbf{Max}\, i \ : \ 0 \leq i < sentn \ : \ S[i] \ ) \\
\Rightarrow & \quad \{ \text{ calculus } \} \\
& sent = (\, \mathbf{Max}\, i \ : \ 0 \leq i < sentn \ : \ S[i] \ )
\end{array}
$$

□

Corollary:

$$delp \wedge delm \geq k \ \mathbf{follows} \ sentp \wedge sent \geq k$$

Proof:

$$
\begin{array}{ll}
& D \ \mathbf{follows}_{\sqsubseteq} \ S \\
\Rightarrow & \quad \{ \ \ \mathbf{max} \ \text{ is monotonic with respect to } \sqsubseteq \ \ \} \\
& (\,\mathbf{Max}\, i \,:\, \mathit{0} \le i < |D| \,:\, D[i]\ ) \ \mathbf{follows} \ \ (\,\mathbf{Max}\, i \,:\, \mathit{0} \le i < |S| \,:\, S[i]\ ) \\
\Rightarrow & \quad \{ \ \text{monotonicity of predicate with respect to } \le \ \} \\
& (\,\mathbf{Max}\, i \,:\, \mathit{0} \le i < |D| \,:\, D[i]\ ) \ge k \ \ \mathbf{follows} \ \ (\,\mathbf{Max}\, i \,:\, \mathit{0} \le i < |S| \,:\, S[i]\ ) \ge k \\
\equiv & \quad \{ \ 4.34\text{:} \ \ \mathit{delp} \ \ \mathbf{follows} \ \ \mathit{sentp} \ \ \} \\
& (\,\mathbf{Max}\, i \,:\, \mathit{0} \le i < |D| \,:\, D[i]\ ) \ge k \ \ \mathbf{follows} \ \ (\,\mathbf{Max}\, i \,:\, \mathit{0} \le i < |S| \,:\, S[i]\ ) \ge k \\
& \ \wedge \ \mathit{delp} \ \ \mathbf{follows} \ \ \mathit{sentp} \\
\Rightarrow & \quad \{ \ \text{conjunctivity of } \ \mathbf{follows} \ \ \} \\
& \mathit{delp} \ \wedge \ (\,\mathbf{Max}\, i \,:\, \mathit{0} \le i < |D| \,:\, D[i]\ ) \ge k \ \mathbf{follows} \\
& \mathit{sentp} \ \wedge \ (\,\mathbf{Max}\, i \,:\, \mathit{0} \le i < |S| \,:\, S[i]\ ) \ge k \\
\equiv & \quad \{ \ 4.26 : \ \mathit{deln} = |D|\,, \text{ and } 4.27 : \ \mathit{sentn} = |S| \ \ \} \\
& \mathit{delp} \ \wedge \ (\,\mathbf{Max}\, i \,:\, \mathit{0} \le i < \mathit{deln} \,:\, D[i]\ ) \ge k \ \mathbf{follows} \\
& \mathit{sentp} \ \wedge \ (\,\mathbf{Max}\, i \,:\, \mathit{0} \le i < \mathit{sentn} \,:\, S[i]\ ) \ge k \\
\equiv & \quad \{ \ 4.38 : \ \mathit{delp} \ \Rightarrow \ \mathit{delm} = (\,\mathbf{Max}\, i \,:\, \mathit{0} \le i < \mathit{deln} \,:\, D[i]\ ) \ \ \} \\
& \mathit{delp} \ \wedge \ \mathit{delm} \ge k \ \ \mathbf{follows} \ \ \mathit{sentp} \ \wedge \ (\,\mathbf{Max}\, i \,:\, \mathit{0} \le i < \mathit{sentn} \,:\, S[i]\ ) \ge k \\
\equiv & \quad \{ \ \text{Lemma 4.9} : \ \mathit{sentp} \ \Rightarrow \ \mathit{sent} = (\,\mathbf{Max}\, i \,:\, \mathit{0} \le i < \mathit{sentn} \,:\, S[i]\ ) \ \ \} \\
& \mathit{delp} \ \wedge \ \mathit{delm} \ge k \ \ \mathbf{follows} \ \ \mathit{sentp} \ \wedge \ \mathit{sent} \ge k
\end{array}
$$

□

Corollary: For ordered channels,

$$\mathit{delp} \Rightarrow \mathit{del} = \mathit{delm}$$

Proof:

$$
\begin{array}{ll}
& \mathit{delp} \\
\equiv & \quad \{ \ 4.28\text{:} \ \ \mathit{delp} \ \equiv \ \mathit{deln} > \mathit{0} \ \ \} \\
& \mathit{deln} > \mathit{0} \\
\equiv & \quad \{ \ 4.37 : \ \ (\forall\, i \,:\, \mathit{0} \le i < \mathit{sentn} - \mathit{1} \,:\, S[i] \le S[i+\mathit{1}]\ ) \ \ \} \\
& \mathit{deln} > \mathit{0} \ \wedge \ (\forall\, i \,:\, \mathit{0} \le i < \mathit{sentn} - \mathit{1} \,:\, S[i] \le S[i+\mathit{1}]\ ) \\
\equiv & \quad \{ \ 4.33 : \ \mathit{deln} \ \ \mathbf{follows} \ \ \mathit{sentn} \ \ \} \\
& \mathit{deln} > \mathit{0} \ \wedge \ (\forall\, i \,:\, \mathit{0} \le i < \mathit{sentn} - \mathit{1} \,:\, S[i] \le S[i+\mathit{1}]\ ) \ \wedge \ \mathit{deln} \le \mathit{sentn} \\
\Rightarrow & \quad \{ \ \text{calculus} \ \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & deln > 0 \;\wedge\; (\forall\, i \,:\, 0 \le i < deln - 1 \,:\, S[i] \le S[i+1]\,) \\
\equiv & \quad \{\ 4.35 : \ D \ \ \mathbf{follows}_{\preceq}\ S\text{, so } (\forall\, i \,:\, 0 \le i < |D| \,:\, D[i] = S[i]\,) \ \} \\
 & deln > 0 \;\wedge\; (\forall\, i \,:\, 0 \le i < deln - 1 \,:\, D[i] \le D[i+1]\,) \\
\Rightarrow & \quad \{\ \text{Lemma 4.8 with k := deln} \ \} \\
 & deln > 0 \;\wedge\; D[deln - 1] = (\,\mathbf{Max}\, i \,:\, 0 \le i < deln \,:\, D[i]\,) \\
\equiv & \quad \{\ \text{4.28: } \ delp \;\equiv\; deln > 0 \ \} \\
 & delp \;\wedge\; D[deln - 1] = (\,\mathbf{Max}\, i \,:\, 0 \le i < deln \,:\, D[i]\,) \\
\equiv & \quad \{\ \text{4.30: } \ delp \;\Rightarrow\; del = D[deln - 1] \ \} \\
 & delp \;\wedge\; del = (\,\mathbf{Max}\, i \,:\, 0 \le i < deln \,:\, D[i]\,) \\
\equiv & \quad \{\ \text{definition of } \ delm \ \} \\
 & delp \;\wedge\; del = delm \\
\Rightarrow & \quad \{\ \text{calculus} \ \} \\
 & del = delm
\end{array}
$$

□

Chapter 5

Certificates and Testing

In this chapter, we examine how certificates can be useful for component testing and debugging. Mappings from the simple certificates introduced at the end of Chapter 3 to run-time checks are given. We discuss the feasibility of the automatic generation of the code for these run-time checks. The practical support for these specification constructs is considered in the context of a real programming language and framework; namely, C++ and the CORBA distributed-object standard.

5.1 Certificates as Assertions

One of the defining characteristics of certificates is locality. A certificate can refer only to local variables, to messages sent by the component, and to messages delivered to the component by the message-passing layer. The intent is that certificates correspond to component specifications, where no requirements are placed on the environment in which the component is placed. They represent both an obligation on the part of the implementor to provide a certain functionality and a promise to the environment that a certain functionality is provided. Since the validity of such certificates depends only on the implementation of the component in question, these certificates can be validated in isolation, before deploying or releasing the component for general use.

In addition to their role as formal specifications of component behavior, certificates can also play an important role in the testing of an implementation. For a component that corresponds to a single implementation object, the predicates of a certificate contain only expressions local to a single implementation object. It is therefore relatively easy to check at run time that the properties described are maintained by the component. When used in this manner, certificates are akin to assertions in sequential programming. They can play a similar role in the debugging phase of software development.

The only requirement on a certificate for it to be run-time checkable is that its predicates all be local. This requirement does not guarantee, however, that the certificate can be locally verified. For example, the certificate **transient**.$hungry.P$ is local to the `Philosopher` component `P`. This certificate can be tested (in the limited manner in which progress properties can be tested, as discussed in Section 5.4). This certificate cannot necessarily, however, be unilaterally guaranteed by the component regardless of the environment in which it is placed (*e.g.*, the transition out of the hungry state may require permision from other components in the system).

5.1.1 When Should Assertions be Checked?

The computational model described in Chapter 2 involves sequences of interleaved atomic actions. Each action maps a system state to a new system state. The certificates described in this chapter involve predicates on these states. The validity of a certificate therefore depends on the state before and after the execution of an atomic action. The intermediate states during the execution of an atomic action are not observable and so are not required to satisfy the certificates.

We associate these atomic actions with the methods that are executed as the result of the delivery of an RPC request. Even though execution of the method may cause several messages to be sent, the entire method is considered to be a single atomic action. It is therefore sufficient to examine the component state at the beginning and the end of these methods. Such an approach is valid under certain constraints on the message-passing layer. A set of constraints sufficient to ensure the validity of this approach has been given in [55]. Our computational model meets these requirements, allowing us to treat methods as atomic blocks.

5.1.2 A Practical Instance of Our Model: CORBA-compliant DSOM

In the remainder of this chapter, we describe how the use of certificates for component testing can be supported in an instantiation of our distributed computing model; namely, the CORBA standard. The CORBA standard is an attractive vehicle for these ideas because component descriptions (with certificates) are entirely consistent with the object declarations of the CORBA IDL. Since CORBA-compliant systems provide IDL parser to generate stub code from these declarations, it is appealing to consider using these parsers to also automatically generate run-time checked assertions from the certificates. These assertions would be embedded in the stub code provided by the IDL parser. In the cases where the code cannot be entirely automatically generated (*e.g.*, see Section 5.2), the required stubs and hooks can be generated and the programmer would then be required to fill in the functionality.

There are many commercially available CORBA-compliant systems. Most of these systems support a variety of implementation languages (usually at least C, C++, Smalltalk, and Java). We will present our run-time translations in the context of IBM's CORBA-compliant SOM/DSOM system, where the component implementations are given in C++. The translation of certificates into run-time checks does not depend on the use of this particular commercial system. No special features of SOM/DSOM are critical to the tractability of the approach outlined in this chapter. This choice was made for illustrative purposes only, as our goal was to demonstrate the practicality of our approach.

5.2 Mapping of Specification Variables

The local variables listed in a component description (and referred to by certificates) do not necessarily correspond directly to implementation variables. Also, the channel histories (and associated functions such as *delp* and *sent*) are not directly available to the implementation in a CORBA application. Thus, a mapping between these specification (or "ghost") variables and the implementation variables must be provided.

The relationship between local variables in the component description and implementation variables must be defined by the implementor. For each variable in the component description, a function is written that calculates the value of the variable from the implementation state. For example, the component description of `Philosopher` contains three boolean local variables: `thinking`, `hungry`, and `eating`. The implementation of this component, however, may contain a single integer variable, `status`, that defines the object's state: 0 when it is thinking, 1 when it is hungry, and 2 when it is eating. For each specification variable, the implementor of the component must provide a function mapping the value of the integer implementation variable to a value for the specification variable. As an example of such a mapping for the local variable `thinking` in a CORBA application written in C++ and using IBM's DSOM, see Program 5.1. The type `PhilosopherData` is a standard data structure used in DSOM to encapsulate the member data of the implementation object. The header of this function can be automatically generated, but the code requires knowledge of the intended relationship between implementation state and specification state.

On the other hand, the relationship between the functions on channel histories (*e.g.*,

```
boolean evaluate_thinking (PhilosopherData *d)
{
  if (d->status == 0)
    return true;
  else
    return false;
}
```

Program 5.1: A mapping from the implementation state to a specification variable.

delp and *sent*) and the implementation channel state is common to all implementations. No user code is required to support these functions, as they can be provided automatically as part of a library. For example, the class given in Program 5.2 records the history of messages delivered on a particular incoming channel. The class is parameterized according to the message type of the channel.

```
template <class MessageType>
struct InChannelHistory  {
    Sequence<MessageType> A;

    int deln (void)  {
      return A.get_length();
    }

    MessageType del (void)  {
      return A.last();
    }

    void update (MessageType m)  {
      A.append(m);
    }
};
```

Program 5.2: A data structure for representing the history of an incoming channel.

As an optimization, the entire channel history need not be preserved. For example, for an outgoing monotonic channel where we care only about the value of *sent*, it is sufficient to record only this (maximum) value. The class given in Program 5.3 implements such an optimization.

Each RPC target is associated with a unique incoming channel history. Every method is modified by the addition of an update of the corresponding channel history. The message delivered to the component is appended to the channel history. For example, consider the

```
template <class MessageType>
struct OutMonotonicChannelHistory  {
    boolean sentp;
    MessageType sent;

    OutMonotonicChannelHistory (void)  {
      sentp = false;
    }

    void update (MessageType m)  {
      assert (m >= sent);
      sentp = true;
      sent = m;
    }
};
```

Program 5.3: An optimized data structure for representing the history of an incoming channel.

method `m` in Program 5.4. The stub for this method, including the signature and the first line (where a variable `somThis` is declared and assigned), is automatically generated by the IDL parser. This method takes three arguments, `x`, `y`, and `a` (`somSelf` and `ev` are handles used by the underlying DSOM system). These arguments correspond to a message type in the specification of the component. The value of the message is constructed from the arguments by a programmer-defined function `evaluate_m_msg()`. Everything but this function and the definition of the message type can be automatically generated from the component's certificates.

Whenever software is used to test software, there are two concerns: efficiency and correctness. The first concern is not critical, as we expect certificate-based testing to occur as part of the debugging of components, and not during their actual use after deployment. For calculations that are prohibitively expensive (for example, see the component specifications in Chapter 8), programmer intervention is certainly required to reduce their complexity. The greater the degree of programmer intervention, however, the greater the issue of correctness becomes a concern. In general, however, the code a programmer is required to provide as part of the testing of a component is relatively simple compared to the complexity of the entire component implementation. Also, this code can be written entirely as traditional sequential code, to which all the verification techniques that have been developed in that area can be applied. We postulate, therefore, that it is easier to write correct code for certificate-based testing than to write correct code for a component implementation.

```
typedef struct  {
  //definition of message type goes here
} m_MessageType;

InChannelHistory<m_MessageType> m_channel;

m_MessageType evaluate_m_msg (long x,
                              short y,
                              ArgStruct *a)
{
  m_MessageType ret_val;
  //implementation of message evaluation goes here
  return ret_val;
}

SOM_Scope void SOMLINK m (Component *somSelf,
                          Environment *ev,
                          long x,
                          short y,
                          ArgStruct *a)
{
  ComponentData *somThis = ComponentGetData(somSelf);

  m_channel.update (evaluate_m_msg(x,y,a));

  //implementation of method m goes here
};
```

Program 5.4: Code to maintain the channel history of an RPC target.

5.3 Support for Safety Properties

5.3.1 Fundamental Safety Certificates

The **initially** certificate describes a predicate that is true at the beginning of the computation. Since no action has had a chance to execute, this predicate must be established at the time of component instantiation. For a component that corresponds to a single implementation object, this certificate must be established by that object's constructor. To test that this certificate holds, then, it is sufficient to test it at the end of the constructor.

The other fundamental safety operator is **next**. To test for the violation of a **next** property, the states before and after an action must be examined. This requires a test both at the beginning of a method and at its end. Notice that the testing at the beginning of the method is meant to capture the state before the action; that is, before the delivery of the RPC request. The channel state must therefore be updated only after this initial examination of state. For a certificate p **next** q and a method `m`, the framework to test for the violation of this certificate is given in Program 5.5. Recall that the signature of this method and the first line (that declares and assigns a variable `somThis`) are both part of the DSOM implementation. They are provided automatically as procedure stubs by the IDL parser. Also, this method does not have any arguments (`somSelf` and `ev` are used by the underlying DSOM system), so a unary value is appended to the channel storing the incoming message history.

```
InChannelHistory m_channel<unary>;
boolean p (ComponentData *);
boolean q (ComponentData *);

SOM_Scope void SOMLINK m (Component *somSelf,
                          Environment *ev)
{
  ComponentData *somThis = ComponentGetData(somSelf);

  boolean pre_predicate = p(somThis);
  m_channel.update (unary::set);

  //implementation of method m goes here

  assert (!pre_predicate || q(somThis));
}
```

Program 5.5: Testing a method for a **next** property.

The situation is complicated somewhat by the generality of the definition of the **next** operator. This definition permits the inclusion of free variables over which the expression can be universally quantified. For example, for a local variable `x` and a free variable `n`, we might have the certificate $x = n$ **next** $x \geq n$. The naive translation of such a quantification, however, yields an infinite number of predicates to be tested before and after the execution of the method.

This complication is avoided in our simple safety certificates of monotonicity, boundedness, unquantified **next**, and functional **next** properties. Because these restricted forms of **next** and **invariant** do not contain free variables (and hence universal quantification), they can be easily (even automatically) mapped onto the skeleton illustrated in Program 5.5.

5.3.2 Monotonicity

The monotonicity of a local variable `x` can be verified by comparing the value of the variable at the beginning and end of each method. Recall that `x` is a specification variable, so its value must be computed from the actual local state using the (programmer-supplied) function `evaluate_x()`. The skeleton for the modification of a generic method `m` is given in Program 5.6.

```
SOM_Scope void SOMLINK m (Component *somSelf,
                          Environment *ev)
{
  ComponentData *somThis = ComponentGetData(somSelf);

  int pre_x_value = evaluate_x (somThis);
  m_channel.update (unary::set);

  //implementation of method m goes here

  assert (pre_x_value <= evaluate_x (somThis));
}
```

Program 5.6: Testing a method for the monotonicity of `x`.

In the case where the monotonic variable does not have a type with an established ordering relation, the less than or equal to operator would have to be written for this class using the usual C++ operator overloading techniques.

5.3.3 Boundedness

A boundedness property is an invariant property. To check that this property is never violated, it must be checked at the end of the constructor and at the end of every method body. For example, consider the certificate that states that local variable `x` is bounded above by a function of local variables `y` and `z`.

$\mathbf{invariant}.(x \le f.(y,z))$

To check this certificate, each method is modified as shown in Program 5.7 (the last line has been added). The additional line is also added to the constructor. The function $f.(y,z)$ is provided by the programmer in the body of the function `evaluate_f()`. The evaluation of the predicate in the certificate is provided by the programmer in the body of the function `evaluate_bound_1()`. (The number 1 is used to distinguish bounds when a component has more than one boundedness certificate.)

```
int evaluate_x (ComponentData *);
int evaluate_y (ComponentData *);
char evaluate_z (ComponentData *);
int evaluate_f (int, int);
boolean evaluate_bound_1 (ComponentData *data)
{
  return (evaluate_x(data) <= evaluate_f(evaluate_y(data),
                                         evaluate_z(data) ));
}

SOM_Scope void SOMLINK m (Component *somSelf,
                          Environment *ev)
{
  ComponentData *somThis = ComponentGetData(somSelf);

  m_channel.update (unary::set);

  //implementation of method m goes here

  assert (evaluate_bound_1(somThis));
}
```

Program 5.7: Testing a method for the boundedness of `x`.

As with monotonicity, a less than or equal to operator can be defined for the type of the bound variables and invoked through overloading.

5.3.4 Unquantified next

An unquantified **next** property is a **next** property that does not contain any free variables. The certificate *eating* **next** *eating* $\vee$ *thinking* is an example of such a property. The predicates in these unquantified certificates can be calculated directly from the component state. Therefore, the framework given in Program 5.5 can be used directly to validate an unquantified **next** property.

5.3.5 Functional next

In a functional **next**, the values of the dummy variables are uniquely determined by the values of the program variables in the prepredicate. Recall that a functional **next** property with dummy variables from the set I and local variables from the set V has the property:

$$(\forall i \,:\, i \in I \,:\, (\exists f \,::\, [p \Rightarrow i = f.V]\,)\,)$$

We introduced the operator **adjacent** and used preprimed variables to indicate values from the previous state. We write **adjacent**.p where p is a predicate on variables in $V \cup {}'V$.

This notation has the benefit of eliminating all dummy variables by replacing them by their functional evaluation using prepredicate variables. To check that the execution of a method satisfies the constraint expressed by this certificate, the values of the preprimed variables are stored at the beginning of the method and then used in the evaluation of the predicate at the end of the method. For a component with a functional **next** certificate with, for example, two preprimed variables, every method `m` would be modified as shown in Program 5.8.

```
boolean p (ComponentData *, int, char);

SOM_Scope void SOMLINK m (Component *somSelf,
                          Environment *ev)
{
  ComponentData *somThis = ComponentGetData(somSelf);

  int pre_x_value = evaluate_x (somThis);
  char pre_y_value = evaluate_y (somThis);
  m_channel.update (unary::set);

  //implementation of method m goes here

  assert (p(somThis, pre_x_value, pre_y_value));
}
```

Program 5.8: Testing a functional **next** property with two preprimed variables.

5.4 Support for Progress Properties

5.4.1 Fundamental Progress Certificate

Unlike safety properties, progress properties cannot be violated by any finite trace. It is therefore not possible to report an error at run time due to the violation of a progress property. It is possible, however, to report a warning when a progress property has failed to be satisfied in a very long time. Although such a warning is not proof of the existence of an error in the implementation of a component, it can be a useful starting point for the examination of an implementation that is believed to be erroneous.

Recall that the fundamental progress property in our model is transience. This certificate states that if the predicate is true, it will eventually become false. For example, recall the example used in Section 3.3 where the `Philosopher` component is in the eating state for a finite length of time (*i.e.*, **transient**.*eating*). The transience of a predicate is monitored by testing it at the end of every method.

The class given in Program 5.9 records the pertinent information for detecting the possible nontransience of a predicate. The time at which the predicate last became true (or the time of the initiation of the computation) is recorded as well as whether or not the predicate is currently true.

```
struct TransientPredicate  {
  boolean holds;
  long time_stamp;
  boolean (*predicate)(ComponentData *);

  void initialize (ComponentData *data)
  {
    holds = (*predicate)(data);
    if (holds)
      time_stamp = get_current_time();
  }

  void update (ComponentData *data)
  {
    boolean b = (*predicate)(data);
    if (!holds && b)
      time_stamp = get_current_time();
    holds = b;
  }
};
```

Program 5.9: A class for recording the history of a transient predicate.

This information can be used to signal a possible error, if the current time exceeds the time stamp of a transient predicate by some threshold debugging value. Alternatively, this information can be used after interruption of execution to determine which transient predicates were true and how long that had been the case. More detailed histories, including which methods had executed since the predicate became true, can also be maintained to further aid in debugging.

The certificate of transience, like the **next** operator, may contain free variables and universal quantification. As with the **next** operator, the presence of universal quantification makes an automatic mapping of a certificate of transience to some program fragment difficult. In practice, however, many **transient** properties of components can be expressed as simple (*i.e.*, unquantified **transient** or functional **transient**) certificates.

5.4.2 Unquantified Transience

An unquantified **transient** property does not contain any free variables. The certificate **transient**.*eating* is an example of such a property. The predicates in these unquantified certificates can be calculated directly from the component state.

In general, for a predicate p , a variable can be declared to store the history of p using the class given above (Program 5.9). This variable is initialized in the component constructor, and then updated at the end of every method. Program 5.10 shows how an unquantified **transient** property can be mapped to the stub code produced by the IDL parser. The method `somDefaultInit()` is the constructor automatically generated by the IDL parser. The only modification to this method required by our run-time certificate checking is the addition of the last two lines (that initialize the transient predicate). Notice that the only code the programmer must enter is an implementation for the function `evaluate_p()`. The rest can be generated automatically.

5.4.3 Functional Transience

Given a component state, there is a single assignment of values to free variables such that the predicate of a functional **transient** property can be true. There are two ways, then, for the transience specified by the certificate to be satisfied:

1. The predicate can cease to be true for any assignment of values to the free variables.
2. The assignment of values to free variables (required for the predicate to be true) can change.

The special syntax of functional **transient** certificates permits a special mapping from these expressions to program fragments. The values of the dummy variables specified in the certificate can be calculated at the end of each method body. Every time the predicate is true and these values are different from the previous values, they are stored along with a time stamp. The difference between the time stamp of the last update and the current time can be used to trigger an alarm that warns the tester that the expression may be failing to be transient. The class used to store the history of a functional **transient** property is shown in Program 5.11.

```
TransientPredicate p;
boolean evaluate_p (ComponentData *data)
{
  //implementation of predicate evaluation goes here
}

SOM_Scope void SOMLINK somDefaultInit (Component *somSelf,
                                       som3InitCtrl* ctrl)
{
  ComponentData *somThis;
  Component_Init_SOMObject_somDefaultInit(somSelf, ctrl);

  //local Component initialization code goes here

  p.predicate = evaluate_p;
  p.initialize (somThis);
}

SOM_Scope void SOMLINK m (Component *somSelf,
                          Environment *ev)
{
  ComponentData *somThis = ComponentGetData(somSelf);

  //implementation of method m goes here

  p.update (somThis);
}
```

Program 5.10: Code to test an unquantified **transient** property.

In general, for a functional transient predicate (with an associated set of dummy variables), a variable can be declared to store the history of this predicate using the class given above. This variable is initialized in the component constructor, and then updated at the end of every method. Program 5.12 shows how a functional **transient** property with a single free variable (of type integer) can be mapped to the stub code produced by the IDL parser. The only coding required of the programmer is an implementation of two functions:

1. The functional calculation of the free variables from the component state.

2. The evaluation of the predicate given the current component state and the values of the free variables.

Notice that a functional **transient** property directly specifies both these functions. In a certificate

$$(\forall i \,:\, i \in I \,:\, i := f_i.V\ )\ \textbf{in}\ \textbf{transient}.p.(I, V)$$

```
template <class SetType>
struct FunctionalTransientPredicate  {
  boolean holds;
  long time_stamp;
  SetType free_vars;
  SetType (*dummies)(ComponentData *);
  boolean (*predicate)(ComponentData *, SetType);

  void initialize (ComponentData *data)
  {
    free_vars = dummies(data);
    holds = (*predicate)(data,free_vars);
    if (holds)
      time_stamp = get_current_time();
  }

  void update (ComponentData *data)
  {
    SetType v = dummies(data);
    boolean b = (*predicate)(data,v);
    if ((!holds && b) || ((v != free_vars) && b))
      time_stamp = get_current_time();
    holds = b;
    free_vars = v;
  }
};
```

Program 5.11: A class for recording the history of a functional transient predicate.

the former is given by the f_i and the latter is given by p .

If there is more than one free variable, a structure is defined that contains all these variables. This structure must contain an implementation of the not equal operator. This data structure, however, is highly regular and can be automatically generated from the certificate. An example of the data structure corresponding to two free variables, one integer and one character, is given in Program 5.13.

This data structure replaces the declaration of `DummyVars` in Program 5.12 (as an integer, in this case). No other code is impacted. The history of the functional **transient** property is initialized and updated as shown in Program 5.12. As before, the programmer must supply functions to calculate the values of the free variables given the component state, and evaluate the predicate given the component state and the values of the free variables.

```
typedef int DummyVars;
FunctionalTransientPredicate<DummyVars> p;

DummyVars calculate_p_dummies (ComponentData *data)
{
  DummyVars ret_val;
  //implementation of calculation of free variable values goes here
  return ret_val;
}

boolean evaluate_p (ComponentData *data, DummyVars d)
{
  //implementation of predicate evaluation goes here
}

SOM_Scope void SOMLINK somDefaultInit (Component *somSelf,
                                       som3InitCtrl* ctrl)
{
  ComponentData *somThis;
  Component_Init_SOMObject_somDefaultInit(somSelf, ctrl);

  //local Component initialization code goes here

  p.dummies = calculate_p_dummies;
  p.predicate = evaluate_p;
  p.initialize (somThis);
}

SOM_Scope void SOMLINK m (Component *somSelf,
                          Environment *ev)
{
  ComponentData *somThis = ComponentGetData(somSelf);

  //implementation of method m goes here

  p.update (somThis);
}
```

Program 5.12: Code to test a functional **transient** property.

```
struct DummyVars  {
  int i;
  char j;

  int operator != (const DummySet& d)
  {
     if ((d.i == i) && (d.j == j))
       return 0;
     else
       return 1;
  }
};
```

Program 5.13: Definition of a class for multiple dummy fariables of a functional **transient** property.

Chapter 6

Services

In this chapter, we introduce services. A service is a frequently used paradigm for component interaction in distributed systems. Two examples of such services are given and their integration in our model is explored. One of these services is then used in the development of a larger example application, illustrating the utility of this approach to paradigm reuse.

6.1 Introduction

The certificate-based approach described in previous chapters emphasizes the specification of the individual component. This specification is given as a unilateral guarantee of component behavior, regardless of the environment in which it is placed. Component properties, then, are verified in isolation of the rest of the system. This promotes the specification, implementation, and proof reuse of the individual components. System properties, on the other hand, are proven as the conjunction of the local component properties. Because of their dependence on all the constituent component properties, the proofs of these system properties are difficult to reuse in different contexts.

All (but the most trivial) system properties are established and maintained through the coordination of component interactions. If these component interactions can be structured in a meaningful way, often the arguments of correctness for the distributed system can be simplified. Many algorithms impose this structuring through a small collection of abstractions that are frequently reused. These abstractions then figure prominently in both the informal exposition of the algorithm as well as its formal verification.

For example, consider a distributed system responsible for maintaining a certain temperature in a building. The system consists of a collection of thermostat components that alternate between two states, off and on. A thermostat changes state depending on the local temperature of its immediate environment: A thermostat is off when the environment temperature is above a certain threshold and it is on when the environment temperature is below a certain threshold. The steady-state of the building temperature then follows directly from the certificates capturing individual component properties. If, however, we impose the further requirement that at most two-thirds of the thermostats can be on at the same time (*e.g.*, perhaps this is required by the electrical system), the thermostats must now coordinate their transitions from off to on and vice-versa. The protocol used for this coordination can be expressed using certificates, but a compositional proof is required to establish that the system invariant is maintained. This chapter addresses the question of how to reduce the burden of this compositional proof.

As mentioned above, one technique for simplifying the proof is to introduce an abstraction that structures the component interactions. The maintenance of a system property is then an immediate consequence of this structuring. Since they are properties of component communication, these abstractions cannot be captured in a single component but must be derived from the specifications of a collection of components. It is desirable, then, to reuse this derivation whenever applicable. A qualitative observation is that a relatively small number of abstractions are used in a relatively large number of distributed algorithms. It is therefore reasonable to expect that extensive reuse is possible. We will call these abstractions "services."

In this chapter we examine two simple services: tokens and logical clocks. The choice of these services is significant. They represent two very powerful mechanisms for structuring component interactions. The former is often used to represent the indivisibility and indestructibility of a limited resource. The latter is often used to establish consistency and causality relationships. It has been our experience that these services reappear frequently in the development of distributed algorithms. For example, they can be used to simplify the proof and exposition of algorithms as varied as mutual exclusion, dining philosophers [23], global snapshots [17], Byzantine generals [81], and termination detection [28, 3].

On the other hand, these two services do not comprise an exhaustive list of useful services. They are given as an illustration of the use of services and their integration in our model. Another example might be partial orders, where components are connected in an acyclic directed graph structure. Edge directions can change but components coordinate these modifications so as to maintain the acyclicity of the graph. This service is useful for the symmetry breaking that is commonly required for fair arbitration. For example, components that are higher in the partial order could be given priority until their request is satisfied, at which point they move to the bottom of the partial order.

We confine our discussion in this chapter, however, to tokens and logical clocks. Each is characterized informally, and their utility in distributed algorithms outlined. Each service is then characterized formally using the certificates given in previous chapters. Since a service represents an abstraction of a communication discipline, we give the certificate-based specification for all the components participating in this discipline. The key properties embodied by these services are then derived from the conjunction of these specifications. We conclude with an extended example, illustrating how a service can be used to simplify the construction of a larger application.

6.2 Tokens

6.2.1 Specification

A token is an indivisible unit that can be in the possession of at most a single component. Thus, the number of components with tokens is bounded above by the number of tokens in the computation. Tokens can neither be created nor destroyed. Thus, the total number of tokens in a computation is constant. In this way, tokens capture a limited resource allocation paradigm: A component with a token has access to the resource, a component without a token does not.

Let the total number of tokens in a system be $tokens$. Let the number of tokens held initially by a component c be $c.initial\_hold$. The first property of tokens is that their total number is constant, and this number is the sum of the number of tokens each component holds initially.

$$\mathbf{invariant}.(tokens = (\,\Sigma\, c :: c.initial\_hold\,)) \tag{6.1}$$

Let the number of tokens held by a component c be $c.holding$ and let the number of tokens in a channel from component c to component c' be $(c, c').holding$. The second property of tokens is that the number of tokens held by any component or any channel is nonnegative.

$$(\,\forall\, c :: \mathbf{invariant}.(c.holding \geq 0)\,) \tag{6.2}$$

$$(\,\forall\, c, c' :: \mathbf{invariant}.((c, c').holding \geq 0)\,) \tag{6.3}$$

Notice that this specification is given entirely in terms of safety properties. There are no guarantees concerning token circulation. These guarantees are protocol dependent and can be layered on top of this fundamental specification. An example of such a layering is given in Section 6.4, where a token manager for enforcing mutual exclusion is developed.

6.2.2 Utility of Tokens

A system invariant that is a conjunction of local component invariants can be derived directly from those component properties. For example, consider the system of thermostat components from the introduction to this chapter. Say each component has a local variable that represents how much energy that component has expended. If each component maintains a local invariant that this energy consumption is bounded above by some value, it follows that the total energy consumption of the entire system is also bounded above by some value (*i.e.*, the sum of the local values).

Tokens, on the other hand, capture a system property that is a disjunction of local component invariants. For example, recall the constraint on the system of thermostats that only two thirds of them can be on at the same time. The system invariant that there exist at most n thermostat components that are on can be captured through the use of tokens: A thermostat component must possess a token in order to be on. By controlling the number of tokens in the system, the number of thermostats that are on is constrained as a consequence.

This constraint is an example of the general problem of mutual exclusion. The problem of mutual exclusion is to control access to a limited resource. Since tokens are indivisible, they are a natural expression of the safety property of mutual exclusion.

6.2.3 Certificate Specification

Tokens are defined to circulate among a collection of components, all of which satisfy the `TokenHolder` specification given in Program 6.1.

```
Component TokenHolder  {
  local const := initial_hold : int       //number of tokens initially held
  local vars :=  holding : int            //number of tokens currently held
  rpc targets := tok(unary) (unordered)
  neighbors :=   N : set of Component TokenHolder
  certificates :=
    Invariant.(holding >= 0)
    Invariant.(holding = initial_hold + (SUMc in N : deln(c,tok))
                         - (SUMc in N : sentn(c,tok)))
}
```

Program 6.1: Description of a `TokenHolder` component.

Each `TokenHolder` component has a local constant, `initial_hold`, that gives the number of tokens held initially by the component, as well as a local variable, `holding`, that gives the number of tokens currently in the component's possession. A token is received from another component when a message is delivered to the `tok()` RPC target. Conversely, a token ceases to be in a component's possession when that component sends a message to another `TokenHolder` component.

6.2.4 Proof of Specification

Prove (6.1):

$$\mathbf{invariant}.(\mathit{tokens} = (\ \Sigma\ c\ ::\ \ c.\mathit{initial\_hold}\ ))$$

Proof:

$$\begin{array}{ll}
 & \mathit{tokens} \\
= & \quad \{\ \text{definition of } \mathit{tokens}\ \} \\
 & (\ \Sigma\ c\ ::\ c.\mathit{holding}\ ) + (\ \Sigma\ c, c'\ ::\ (c, c').\mathit{holding}\ ) \\
= & \quad \{\ \text{definition of } (c, c').\mathit{holding}\ \} \\
 & (\ \Sigma\ c\ ::\ c.\mathit{holding}\ ) + (\ \Sigma\ c, c'\ ::\ \mathit{sentn}(c, c', \mathit{tok}) - \mathit{deln}(c, c', \mathit{tok})\ ) \\
= & \quad \{\ \mathbf{invariant}.(c.\mathit{holding} = c.\mathit{initial_hold} + (\ \Sigma\ c'\ ::\ \mathit{deln}(c', c, \mathit{tok})\ ) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad - (\ \Sigma\ c'\ ::\ \mathit{sentn}(c, c', \mathit{tok})\ ))\ \ \} \\
 & (\ \Sigma\ c\ ::\ c.\mathit{initial_hold} + (\ \Sigma\ c'\ ::\ \mathit{deln}(c', c, \mathit{tok})\ ) - (\ \Sigma\ c'\ ::\ \mathit{sentn}(c, c', \mathit{tok})\ )\ ) \\
 & +\,(\ \Sigma\ c, c'\ ::\ \mathit{sentn}(c', c, \mathit{tok}) - \mathit{deln}(c', c, \mathit{tok})\ ) \\
= & \quad \{\ \text{calculus}\ \} \\
 & (\ \Sigma\ c\ ::\ c.\mathit{initial_hold}\ )
\end{array}$$

□

Prove (6.2):

$$(\forall\, c\ ::\ \mathbf{invariant}.(c.\mathit{holding} \geq 0)\ )$$

Proof: This follows directly from the certificates of the `TokenHolder` component.
□

Prove (6.3):

$$(\forall\, c, c'\ ::\ \mathbf{invariant}.((c, c').\mathit{holding} \geq 0)\ )$$

Proof:

$$
\begin{array}{ll}
 & (c,c').holding \\
= & \quad \{ \text{ definition of } (c,c').holding \quad \} \\
 & sentn(c,c',tok) - deln(c,c',tok) \\
\geq & \quad \{ \text{ channel property } \} \\
 & 0
\end{array}
$$

□

6.3 Logical Clocks

6.3.1 Specification

A logical clock is a monotonically nondecreasing counter maintained by a component. Each component maintains its own logical clock. The key property of the interaction of clocked components is that all messages exchanged are time stamped. An outgoing message is time stamped with the value of the logical clock of the component sending the message. Conversely, the time stamp of an incoming message is used to update the logical clock of the component receiving the message. The logical clock of the component receiving the message is set to a value greater than the time stamp of the incoming message (while maintaining its monotonicity).

Logical clocks are a partial encoding of causality. More specifically, logical clocks disallow certain chains of causality between components. Since components can interact only by messages, this restriction reflects a requirement on the messages that components exchange. The fundamental system property resulting from the use of the logical clock protocol is that messages are not received at an earlier logical time than when they are sent. That is, a component with a logical clock value of n has not received any messages that were sent after a logical (*i.e.*, local to the sending component) time of n .

The diagram in Figure 6.1 illustrates the use of logical clocks. The history of actions at a component is represented by a vertical time line, while a message exchanged between components is represented by a dashed arrow. The key system property is reflected in the observation that all dashed arrows are directed upwards (*i.e.*, messages are received later than they are sent).

With each component, x , we associate a history, $x.H$, of actions. This history is a sequence of tuples. Each tuple consists of the entire state of the component, including the logical time and the number of messages sent and delivered. For example, $x.H[i].time$ is

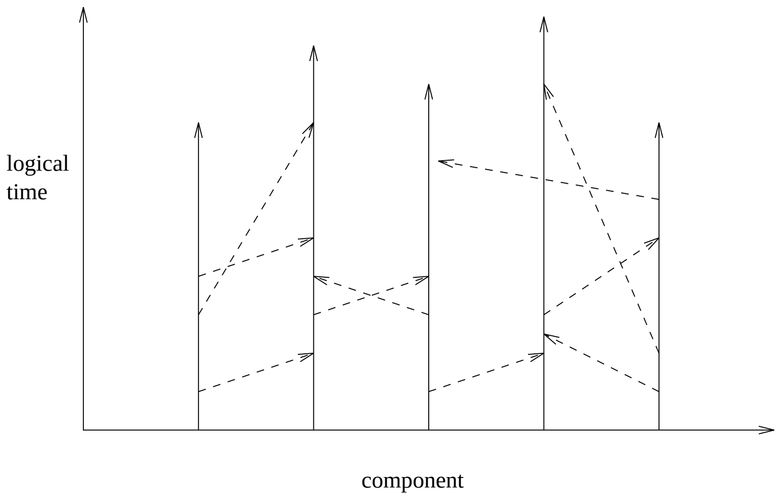

Figure 6.1: Graphical time line for a collection of components exchanging messages.

the logical time at the i^{th} tuple. To simplify the notation, we consider components to have a single channel between them (this allows channels to be identified by the sending and receiving component, without mentioning the RPC target). The system property that no messages are delivered at an earlier logical time than when they were sent is expressed by the following equation:

$$\begin{aligned}&\mathbf{invariant}.(x.H[i].time = t \wedge x.H[i].deln(y,x) = k \Rightarrow \\ &\qquad (\forall j \; : \; y.H[j].time \geq t \; : \; y.H[j].sentn(y,x) \geq k \;)) \qquad (6.4)\end{aligned}$$

6.3.2 Utility of Logical Clocks

Logical clocks reflect a simple causality relationship. As discussed above, the use of logical clocks entails a discipline of time stamping outgoing messages and of updating the logical clock according to incoming messages. As a result of this discipline, a component with a logical time of n has not received any messages sent at a logical time greater than n.

One of the useful consequences of this system property is that a valid global snapshot [19, Chapter 10] can be recorded quite easily. It is sufficient for all components to record their state when their logical clocks have reached a particular value. Global snapshots are especially useful for the detection of **stable** properties (such as termination or deadlock).

Logical clocks can also be useful in algorithms that rely on a total ordering of actions. The value of a component's logical clock is used as the basis of this total ordering (with ties broken in any arbitrary manner). For example, a mutual exclusion algorithm might be based on granting access to the critical section to components in the order in which their requests were made (in logical time) [53, 85]. The fact that logical time advances by some positive amount guarantees that all components are eventually granted access to the critical section.

6.3.3 Certificate Specification

In a system that uses logical clocks, each component must satisfy the `Clocked` specification given in Program 6.2.

```
Component Clocked  {
  local vars :=  time : int                //local time
                 H : sequence of tuples    //H[t].time, deln(x), sentn(x)
  neighbors :=   N : Component Clocked
  certificates :=
    Stable.(sentp(c) ^ sent(c) >= k)
    Stable.(time >= k)
    Transient.(time = k)
    Invariant.(Ac in N : delp(c) : time > del(c))
    sentn(c) = k ^ time = t  Next
        sentn(c) = k v (sentn(c) = k+1 ^ sent(c) = t)
    Invariant.(sentn(c) >= k  ==>
               (Ei : H[i].time = S[k-1] ^ H[i].sentn(c) = k) )
}
```

Program 6.2: Description of a `Clocked` component.

Each `Clocked` component has a local variable for the logical time (`time`) and a local variable for its associated history (`H`). The values a component sends to its neighbors' RPC targets are monotonically nondecreasing, as is its logical time. On the other hand, a component's logical time is guaranteed to change (*i.e.*, increase). A component's logical time is greater than the time stamp of any message delivered to it. Messages sent are time stamped with a component's current logical time. Finally, the history sequence is updated at least frequently enough to reflect each individual send action. That is, for every send action performed by the component, there is a tuple in the history with the time of the time stamp of that message.

6.3.4 Proof of Specification

Prove (6.4):

$$\textbf{invariant}.(x.H[i].time = t \wedge x.H[i].deln(y,x) = k \Rightarrow$$
$$(\forall j \ : \ y.H[j].time \geq t \ : \ y.H[j].sentn(y,x) \geq k \ ))$$

Proof: The specification is trivially true for $k = 0$. So consider $k > 0$:

$$
\begin{array}{ll}
 & x.H[i].time = t \ \wedge \ x.H[i].deln(y,x) = k \\
\equiv & \quad \{ \text{ assumption : } k > 0 \ \} \\
 & x.H[i].time = t \ \wedge \ x.H[i].deln(y,x) = k \ \wedge \ x.H[i].deln(y,x) > 0 \\
\equiv & \quad \{ \text{ channel property } \} \\
 & x.H[i].time = t \ \wedge \ x.H[i].deln(y,x) = k \ \wedge \ x.H[i].delp(y,x) \\
\Rightarrow & \quad \{ \text{ property : } \textbf{invariant}.(H[i].time.delp(c) \ \Rightarrow \ H[i].time > H[i].del(c)) \ \} \\
 & t > x.H[i].del(y,x) \ \wedge \ x.H[i].deln(y,x) = k \\
\equiv & \quad \{ \text{ channel property : } H[i].del(c) = D(c)[H[i].deln(c) - 1] \ \} \\
 & t > D(y,x)[k-1] \ \wedge \ x.H[i].deln(y,x) = k \\
\Rightarrow & \quad \{ \ deln \text{ is monotonic } \} \\
 & deln(y,x) \geq k \ \wedge \ D(y,x)[k-1] < t \\
\Rightarrow & \quad \{ \text{ channel properties } \} \\
 & sentn(y,x) \geq k \ \wedge \ S(y,x)[k-1] < t \\
\Rightarrow & \quad \{ \text{ property : } \textbf{invariant}.(sentn(c) \geq k \ \Rightarrow \ (\exists i \ :: \ H[i].time = S(c)[k-1] \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \ H[i].sentnc = k \ )) \ \} \\
 & (\exists i \ :: \ y.H[i].time = S(y,x)[k-1] \ \wedge \ y.H[i].sentny, x = k \ ) \\
 & \ \wedge \ S(y,x)[k-1] < t \\
\Rightarrow & \quad \{ \text{ calculus } \} \\
 & (\exists i \ :: \ y.H[i].time < t \ \wedge \ y.H[i].sentn(y,x) = k \ ) \\
\Rightarrow & \quad \{ \text{ both } time \text{ and } sentn \text{ are monotonic } \} \\
 & (\forall j \ : \ y.H[j].time \geq t \ : \ y.H[j].sentn(y,x) \geq k \ )
\end{array}
$$

□

6.4 Example: Central Token Manager

In this section we illustrate how the services introduced above can be used in the development of a larger application. In particular, we develop a solution to the mutual exclusion problem based on a central token manager.

6.4.1 Mutual Exclusion with Tokens

The task is to provide mutually exclusive access to a critical section for a collection of components. We solve this problem by requiring that a component possess a token before entering the critical section. By virtue of the token service described in Section 6.2, the number of components in the critical section is bound by a constant (the number of tokens in the system). The system consists of an arbitrary number of `TokenClient` components and a single `TokenManager` (responsible for controlling access to the critical section). Let C be the set of `TokenClient` components and let M be the `TokenManager`. The topology of connections for this system is illustrated in Figure 6.2.

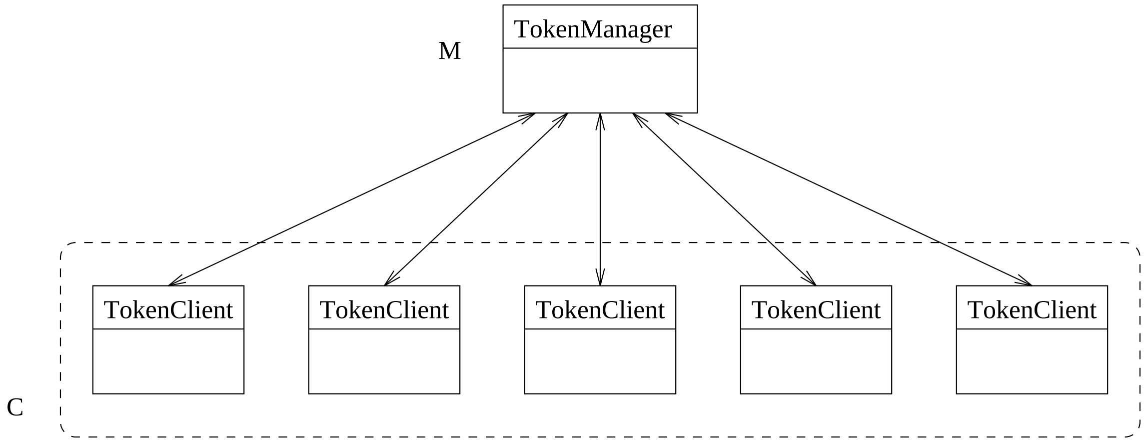

Figure 6.2: Topology of the central token manager solution for mutual exclusion.

When a `TokenClient` wishes to enter the critical section, it sends a request to the `TokenManager`. When it receives a token in reply, the `TokenClient` can enter the critical section. Upon exiting the critical section, the `TokenClient` must return the token to the `TokenManager`. It is the responsibility of every `TokenClient` to eventually exit the critical section (and release the token). It is the responsibility of the `TokenManager` to satisfy token requests in a fair manner (*i.e.*, every `TokenClient` that sends a request eventually gets a token).

One of the required properties of this system is that the number of `TokenClient` components in the critical section is bounded above by a constant, N. A `TokenClient` component c is in the critical section exactly when the boolean $c.critical$ is true.

$$\textbf{invariant}.((\#\, c \;:\; c \in C \;:\; c.critical\,) \leq N) \tag{6.5}$$

The progress property for the system is that token requests are eventually satisfied.

$$(\forall c \ : \ c \in C \ : \ sentn(c, M, req) \geq k \leadsto deln(M, c, tok) \geq k \ ) \tag{6.6}$$

6.4.2 Component Specifications

Initially, the `TokenManager` holds all the tokens. The `TokenManager` maintains a queue of components with outstanding requests. It has two RPC targets, `req()` to receive token requests and `tok()` to receive tokens. The component description of `TokenManager` is given in Program 6.3.

```
Component TokenManager  {
  local const := initial_hold : int        //tokens held initially
  local vars :=  R : queue of Component TokenClient
                                           //clients with pending requests
                 holding : int             //number of tokens held by manager
  rpc targets := req(unary) (unordered)
                 tok(unary) (unordered)
  neighbors :=   C : set of Component TokenClient
  certificates :=
    Invariant.(initial_hold = N)
    Invariant.(holding >= 0)
    Invariant.(holding = initial_hold + (SUMc in C : deln(c,tok))
                         - (SUMc in C : sentn(c,tok)))
    Invariant.(sentn(c,tok) <= deln(c,req))
    Invariant.(sentn(c,tok) < deln(c,req)  ==>  !empty.R ^ (Ei : R[i] = c))
    Invariant.(!empty.R  ==> (Ec in C : sentn(c,tok) > deln(c,tok)))
}
```

Program 6.3: Description of the `TokenManager` component.

A `TokenClient`, on the other hand, initially holds no tokens. It has a boolean variable, `critical`, that is true exactly when the component is in the critical section. A `TokenClient` does not enter the critical section without holding a token. Also, a `TokenClient` eventually returns all tokens that are sent to it. The component description of `TokenClient` is given in Program 6.4.

Notice that both `TokenManager` and `TokenClient` satisfy the `TokenHolder` component specification given in Section 6.2.

6.4.3 Proof of Solution

Prove (6.5): Tokens are constant.

```
Component TokenClient  {
  local const := initial_hold : int        //tokens held initially
  local vars :=  holding : int             //number of tokens held by clients
                 critical : boolean        //is component in critical section?
  rpc targets := tok(unary) (unordered)
  neighbors :=   M : Component TokenManager
  certificates :=
    Invariant.(initial_hold = 0)
    Invariant.(holding = deln(M,tok) - sentn(M,tok))
    sentn(M,tok)  Follows  deln(M,tok)
    Invariant.(critical ==> holding > 0)
}
```

Program 6.4: Description of the `TokenClient` component.

Proof: Follows from all components satisfying the `TokenHolder` specification.
□

Prove (6.6):

$$(\forall c \ : \ c \in C \ : \ \mathit{sentn}(c, M, \mathit{req}) \geq k \leadsto \mathit{deln}(M, c, \mathit{tok}) \geq k \ )$$

Proof:

$$\begin{array}{ll} & \mathit{sentn}(c, M, \mathit{req}) \geq k \\ \leadsto & \quad \{ \text{ channel property } \} \\ & \mathit{deln}(c, M, \mathit{req}) \geq k \\ \leadsto & \quad \{ \text{ see below } \} \\ & \mathit{sentn}(M, c, \mathit{tok}) \geq k \\ \leadsto & \quad \{ \text{ channel property } \} \\ & \mathit{deln}(M, c, \mathit{tok}) \geq k \end{array}$$

So, we must prove:

$$(\forall c \ : \ c \in C \ : \ \mathit{deln}(c, M, \mathit{req}) \geq k \leadsto \mathit{sentn}(M, c, \mathit{tok}) \geq k \ )$$

This property is established by induction from the following property:

$$\mathit{deln}(c, M, \mathit{req}) > \mathit{sentn}(M, c, \mathit{tok}) \wedge \mathit{sentn}(c, M, \mathit{req}) \geq k$$
$$\leadsto \mathit{sentn}(M, c, \mathit{tok}) \geq k + 1$$

$$
\begin{array}{ll}
& deln(c, M, req) > sentn(M, c, tok) \ \wedge \ sentn(c, M, req) \geq k \\
\Rightarrow & \quad \{ \text{ component property: } sentn(M, c, tok) < deln(c, M, req) \ \Rightarrow \\
& \qquad\qquad\qquad\qquad\qquad \neg empty.R \ \wedge \ (\exists\, i :: \ R[i] = c \ ) \ \} \\
& \neg empty.R \ \wedge \ (\exists\, i \ : \ 0 \leq i < |R| \ : \ R[i] = c \ ) \ \wedge \ sentn(c, M, req) \geq k \\
\leadsto & \quad \{ \text{ Aside 1 } \} \\
& sentn(M, c, tok) \geq k + 1
\end{array}
$$

□

Aside 6.1

$$
\begin{array}{ll}
& \neg empty.R \wedge (\exists\, i \ : \ 0 \leq i < |R| \ : \ R[i] = c \ ) \wedge sentn(c, M, req) \geq k \\
\leadsto & sentn(M, c, tok) \geq k + 1
\end{array}
$$

Proof: The result follows by induction from:

$$\neg empty.R \wedge R[0] = c \wedge sentn(M, c, tok) \geq k \leadsto sentn(M, c, tok) \geq k + 1 \tag{6.7}$$

$$(\forall\, i \ : \ 0 < i < |R| \ : \ \neg empty.R \wedge R[i] = c \leadsto \neg empty.R \wedge R[i-1] = c \ ) \tag{6.8}$$

Both 6.7 and 6.8 follow from Lemma 6.1.

□

Lemma 6.1

$$
\begin{array}{ll}
& \neg empty.R \wedge R[0] = c \wedge R = R' \wedge sentn(M, c, tok) \geq k \\
\leadsto & R = tail.R' \wedge sentn(M, c, tok) \geq k + 1
\end{array}
$$

Proof: We make use of the following progress-safety-progress (PSP) theorem in the proof of this lemma:

$$(p \leadsto q) \wedge (r \ \mathbf{next} \ s) \Rightarrow (p \wedge r \leadsto (q \wedge r) \vee (\neg r \wedge s))$$

In particular, we use the following variation of PSP:

$$(r \leadsto q) \wedge (r \ \mathbf{next} \ r \vee s) \wedge \mathbf{invariant}.(\neg(q \wedge r)) \Rightarrow (r \leadsto s)$$

The lemma is proven by the application of this PSP variation with the following substitutions:

$$
\begin{array}{lll}
r & := & \neg empty.R \wedge R[0] = c \wedge R = R' \\
& & \wedge \ (\forall\, i \ : \ i \in C \ : \ deln(i, M, tok) = n_i \ ) \wedge sentn(M, c, tok) \geq k
\end{array} \tag{6.9}
$$

$$q \ := \ (\exists\, i \ : \ i \in C \ : \ deln(i, M, tok) > n_i \ ) \tag{6.10}$$

$$s \ := \ R = tail.R' \wedge sentn(M, c, tok) \geq k + 1 \tag{6.11}$$

Thus, we must establish the leads-to, **next** , and **invariant** properties required by the PSP variation. The **next** property follows immediately from the certificates of the `TokenManager` component. The **invariant** property follows immediately from predicate calculus. Only the leads-to property remains as a proof obligation.

$$
\begin{array}{ll}
 & \neg empty.R \ \wedge \ R[0] = c \ \wedge \ R = R' \\
 & \ \wedge \ (\forall\, i \ : \ i \in C \ : \ deln(i, M, tok) = n_i \ ) \ \wedge \ sentn(M, c, tok) \geq k \\
\Rightarrow & \qquad \{ \ \text{calculus} \ \} \\
 & \neg empty.R \ \wedge \ (\forall\, i \ : \ i \in C \ : \ deln(i, M, tok) = n_i \ ) \\
\Rightarrow & \qquad \{ \quad \textbf{invariant}.(\ \neg empty.R \ \Rightarrow \\
 & \qquad\qquad\qquad\qquad (\exists\, c \ : \ c \in C \ : \ sentn(M, c, tok) > deln(c, M, tok) \ )) \quad \} \\
 & (\exists\, c \ : \ c \in C \ : \ sentn(M, c, tok) > deln(c, M, tok) \ ) \\
 & \ \wedge \ (\forall\, i \ : \ i \in C \ : \ deln(i, M, tok) = n_i \ ) \\
\leadsto & \qquad \{ \ \text{channel property:} \ \ (\forall\, c \ :: \ deln(c, M, tok) \ \ \textbf{follows} \ \ sentn(M, c, tok) \ ) \quad \} \\
 & (\exists\, i \ : \ i \in C \ : \ deln(i, M, tok) > n_i \ )
\end{array}
$$

□

Chapter 7

Example: A Distributed Auction

In this chapter, we present an example that illustrates the utility of certificates in the verification phase of the development of a distributed application. An operational overview of the problem and solution is presented, followed by a formal specification of the system and of the individual components using certificates. A formal proof is given that the composition of these components meets the required system specification.

The application considered is an example of a reactive (rather than transformational) system. That is, we are interested not only in the final solution, but also in the computational path by which that solution is reached.

7.1 The Problem

We wish to design a system to support a multiparty distributed auction. The aim of the auction is to sell a particular item to the bidder willing to pay the highest price.

There are therefore two things to be determined: the identity of the winning bidder and the price of the item. The former is uniquely determined by each bidder's interest in the item and financial resources. The latter, however, is nondeterministic. The only requirement on the sale price is that it be fair to both the bidders and the auctioneer. That is, the final sale price must be low enough that the winning bidder can afford it, yet be high enough that no other bidder can beat it.

Another constraint on the system concerns the manner in which the auctioneer is permitted to determine this fair sale price. We disallow a solution requiring all bidders to communicate their maximum bid to the auctioneer (which could then select the winner and set a price between this winner's maximum bid and the next highest maximum bid). Such a solution requires bidders to trust the auctioneer in a manner that is not realistic in a commercial setting. We also disallow collusion on the part of the bidders to preselect the winner.

The auctioneer is permitted to set a minimum price, below which the item will not be sold.

7.2 A Solution

We implement here a solution with a single centralized auctioneer and a distributed collection of some arbitrary number of bidding processes. The participating bidders communicate only with the auctioneer. This topology reflects the physical distribution of participants in the auction. A hierarchical approach is also possible and would have scalability benefits, but it is not pursued here.

The auctioneer begins by announcing the starting price to all participating bidders. If all bidders decline to place a bid, the auction terminates without a sale. Otherwise, the auctioneer updates the price as bids arrive. When the current price is beaten by a bid, the new price is announced to all bidders other than the bidder placing this high bid. This process is repeated until all the bidders receiving a new price announcement decline to bid. The bidder with the remaining high bid is the winner and the current price is the final sale price.

Each bidder has a maximum value it is willing to bid. This value may be limited by the bidder's financial resources or interest in the item being sold. This maximum value is considered to be constant for the auction. The bidder receives announcements of the current price from the auctioneer. If the current price is greater than or equal to the bidder's maximum bid, it replies by declining to bid.

On the other hand, if the current price is less than the maximum bid, the bidder may reply with a higher bid. The value of the bid submitted is nondeterministically chosen from the interval greater than the current price, but less than or equal to the bidder's maximum bid. Also, this reply need not be immediate. This permits human interaction, where a human bidder must decide the value of bid to submit.

Like a traditional auction, there is no guarantee that the winning bidder pays the minimum fair price (that is, the price equal to the second highest maximum bid among the participants). No bidding strategy can guarantee that the winner will buy the item for this minimum fair price, since this bid may be placed by the bidder with the second highest maximum bid, requiring the winner to bid more.

The bidder state of having received a price announcement that it can beat is a mixed node [33] in the bidder protocol. That is, from this state either a message can be received by the bidder (*i.e.*, a new price announcement) or a message can be sent by the bidder (*i.e.*, a competitive bid). This behavior arises from the bidder's use of a clock to generate a time-out signal, as discussed in Section 3.3.2.

7.3 Auctioneer and Bidder Components

Having described operationally the functionality of the auctioneer and bidders, we now present the component descriptions of these objects. The certificates for each component are given separately from the rest of the component description for readability.

7.3.1 Auctioneer

The auctioneer accepts two types of messages: integer (used to encode a bid) and unary (used to decline further bidding). There is a single constant, `Start`, representing the initial offering price of the item. In addition, there are three variables: `price` (the current highest bid), `winner` (the bidder with the current highest bid), and `sold` (whether or not the auction has completed with a sale). The auctioneer has two RPC targets: `bid` and `nobid`. The former is used to submit a bid, and the latter is used to decline further participation in the auction. Finally, as neighbors, the auctioneer maintains `B`, a set of `Bidder` components. This component description is summarized in Program 7.1.

```
Component Auctioneer  {
  local const := Start : int
  local vars :=  price : int
                 winner : Component Bidder
                 sold : boolean
  rpc targets := bid (int) (monotonic)
                 nobid (unary) (write-once)
  neighbors :=   B : set of Component Bidder
}
```

Program 7.1: Description of the `Auctioneer` component.

7.3.2 Bidder

A bidder accepts a single type of message, an integer representing the current highest bid. There is a single constant for each bidder, `MaxBid`, representing the maximum amount this bidder is willing to pay for the item. A bidder has a single RPC target, `newprice`, used to announce the current highest bid. Finally, each bidder has a single neighbor, the auctioneer (`A`). This component description is summarized in Program 7.2.

```
Component Bidder  {
  local const := MaxBid
  rpc targets := newprice (int) (monotonic)
  neighbors :=   A : Component Auctioneer
}
```

Program 7.2: Description of the `Bidder` component.

7.3.3 Certificates

In this section we give the certificate-based specification of the `Auctioneer` and `Bidder` components introduced above. We use the abbreviation "*np*" to denote the RPC target `newprice()` of the `Bidder` component.

Auctioneer

The following certificates characterize the behavior of the `Auctioneer` component (where x is understood to range over all elements of `B`, the set of participating bidders):

- The price is monotonically increasing.

$$price = n \ \mathbf{next} \ price \geq n$$

- The price is bounded below by the initial offer price.

$$\mathbf{invariant}.(price \geq Start)$$

- The price increases above the initial offer price only if it has been bid by a component (designated the winner). All other components are sent an announcement of this new price.

$$\begin{aligned}\mathbf{invariant}.(price > Start \Rightarrow\ & delp(winner, bid) \wedge del(winner, bid) = price \\ & \wedge\ sentp(winner, np) \wedge Start \leq sent(winner, np) < price \\ & \wedge\ (\forall x \,:\, x \neq winner \,:\, sentp(x, np) \wedge sent(x, np) = price\ ))\end{aligned}$$

- When the price is the initial offer price, all participating bidders are sent an announcement with this price.

$$\textbf{invariant}.(\mathit{price} = \mathit{Start} \Rightarrow (\forall x :: \mathit{sentp}(x, \mathit{np}) \wedge \mathit{sent}(x, \mathit{np}) = \mathit{price}))$$

- The price is bounded below by all the bids that have been received.

$$\textbf{invariant}.((\forall x : \mathit{delp}(x, \mathit{bid}) : \mathit{price} \geq \mathit{del}(x, \mathit{bid})))$$

- The item is sold exactly when all bidders apart from the winner have declined to bid and the price is greater than the initial offer price.

$$\begin{aligned}\textbf{invariant}.(\mathit{sold} \equiv\ & (\forall x : x \neq \mathit{winner} : \mathit{delp}(x, \mathit{nobid})) \\ & \wedge \neg \mathit{delp}(\mathit{winner}, \mathit{nobid}) \wedge \mathit{price} > \mathit{Start})\end{aligned}$$

Bidder

The following certificates characterize the behavior of the `Bidder` component:

- A bidder declines further participation in the auction exactly when a price is announced that is greater than or equal to the bidder's maximum bid.

$$\textbf{invariant}.(\mathit{delp}(A, \mathit{np}) \wedge \mathit{del}(A, \mathit{np}) \geq \mathit{MaxBid} \equiv \mathit{sentp}(A, \mathit{nobid}))$$

- A bidder eventually responds if the announced price is below the bidder's maximum bid.

$$\begin{aligned}\textbf{transient}.(& n < \mathit{MaxBid} \wedge \mathit{delp}(A, \mathit{np}) \wedge \mathit{del}(A, \mathit{np}) = n \\ & \wedge \neg(\mathit{sentp}(A, \mathit{bid}) \wedge \mathit{sent}(A, \mathit{bid}) > n))\end{aligned}$$

- A bidder does not bid more than its maximum bid.

$$\textbf{invariant}.(\mathit{sentp}(x, A, \mathit{bid}) \Rightarrow \mathit{sent}(x, A, \mathit{bid}) \leq x.\mathit{MaxBid})$$

7.4 Proof of Correctness

7.4.1 Problem Specification

A rigorous proof of correctness must have a rigorous specification. We therefore restate the informal specification given in Section 7.1 in more precise terms.

We use A to denote the single `Auctioneer` component in the system, and the variable x to range over all `Bidder` components participating in the auction.

- If possible, there is a winner. This progress condition states that if there is a bidder that can afford the initial offer price, the auction eventually terminates with a sale.

$$(\exists x :: A.Start < x.MaxBid) \leadsto A.sold \tag{7.1}$$

- If no winner is possible, the item is declined. This progress condition states that if there is no bidder that can afford the initial offer price, the auction eventually terminates without a sale.

$$(\forall x :: A.Start \geq x.MaxBid) \leadsto (\forall x :: delp(x, A, nobid)) \tag{7.2}$$

- The selling price is fair to the auctioneer. If an item sells, the selling price could not be beaten by any bidder other than the winner.

$$\textbf{invariant}.(A.sold \Rightarrow (\forall x : x \neq A.winner : A.price \geq x.MaxBid)) \tag{7.3}$$

- The winner can afford to purchase the item. If an item sells, the winner of the auction can afford the selling price.

$$\textbf{invariant}.(A.sold \Rightarrow A.winner.MaxBid \geq A.price) \tag{7.4}$$

- Bidding an amount is a commitment to pay that amount if the bid is accepted. This requirement excludes solutions in which all bidders transmit their `MaxBid` value and the auctioneer picks the winner and a price between the two greatest bids received. Such a solution is not acceptable in a scenario where the auctioneer is not implicitly trusted by the bidders.

$$sentp(x, A, bid) \wedge sent(x, A, bid) \geq n \leadsto A.price \geq n \tag{7.5}$$

7.4.2 Composition of Auctioneer and Bidder Specifications

The composition of these components to form our distributed auction consists of the binding of the neighbor values of the participating components. The set `B` in the `Auctioneer` component is assigned the set of participating `Bidder` components. Conversely, the neighbor value `A` of each bidder is assigned the single `Auctioneer` component in the system.

Recall the simple compositional rule for certificates: A certificate of a component is also a certificate of any system of which that component is a part. Thus, the certificates given in Section 7.3.3 are also certificates for the entire system. We repeat these certificates here because, unlike their presentation in Section 7.3.3, the identification of channels is no longer clear from context, but must be made explicit. Also, as an abbreviation, we will omit the keyword **invariant**. Our convention is that properties that are not **next**, stable, **transient**, or leads-to, are understood to be invariant.

Auctioneer

$$A.price = n \ \textbf{next}\ A.price \geq n \tag{7.6}$$

$$A.price \geq A.Start \tag{7.7}$$

$$\begin{aligned} &A.price > A.Start \Rightarrow delp(A.winner, A, bid) \wedge del(A.winner, A, bid) = A.price \\ &\qquad \wedge\ sentp(A, A.winner, np) \wedge A.Start \leq sent(A, A.winner, np) < A.price \\ &\qquad \wedge\ (\forall x : x \neq A.winner : sentp(A, x, np) \wedge sent(A, x, np) = A.price\ ) \end{aligned} \tag{7.8}$$

$$A.price = A.Start \Rightarrow (\forall x :: sentp(A, x, np) \wedge sent(A, x, np) = A.price\ ) \tag{7.9}$$

$$(\forall x : delp(x, A, bid) : A.price \geq del(x, A, bid)\ ) \tag{7.10}$$

$$\begin{aligned} &A.sold \equiv (\forall x : x \neq A.winner : delp(x, A, nobid)\ ) \\ &\qquad \wedge \neg delp(A.winner, A, nobid) \wedge A.price > A.Start \end{aligned} \tag{7.11}$$

From these equations (in particular (7.7), (7.8), and (7.9)), we can immediately derive the following corollaries:

$$sentp(A, x, np) \wedge sent(A, x, np) \leq A.price \tag{7.12}$$

$$sentp(A, x, np) \wedge sent(A, x, np) \geq A.Start \tag{7.13}$$

$$(\forall x : x \neq A.winner : sentp(A, x, np) \wedge sent(A, x, np) = A.price\ ) \tag{7.14}$$

Bidders

$$delp(A, x, np) \wedge del(A, x, np) \geq x.MaxBid \equiv sentp(x, A, nobid) \tag{7.15}$$

$$\begin{aligned} &\textbf{transient}.(n < x.MaxBid \wedge delp(A, x, np) \wedge del(A, x, np) = n \\ &\qquad\qquad \wedge \neg(sentp(x, A, bid) \wedge sent(x, A, bid) > n)) \end{aligned} \tag{7.16}$$

$$sentp(x, A, bid) \Rightarrow sent(x, A, bid) \leq x.MaxBid \tag{7.17}$$

From these properties (in particular (7.16)), we derive the following corollary:

$$\begin{aligned} &n < x.MaxBid \wedge delp(A, x, np) \wedge del(A, x, np) = n \leadsto \\ &\quad (delp(A, x, np) \wedge del(A, x, np) > n) \vee (sentp(x, A, bid) \wedge sent(x, A, bid) > n) \end{aligned} \tag{7.18}$$

7.4.3 Proof of Solution

We begin with a few lemmas which are helpful in establishing the result.

Lemma 7.1

$$(\,\exists\, x \;::\; \mathit{delp}(A, x, np) \wedge \mathit{del}(A, x, np) > n\;) \Rightarrow A.\mathit{price} > n$$

Proof:

$$
\begin{array}{ll}
 & (\,\exists\, x \;::\; \mathit{delp}(A, x, np) \;\wedge\; \mathit{del}(A, x, np) > n\;) \\
\Rightarrow & \quad \{\ \text{channel properties}\ \} \\
 & (\,\exists\, x \;::\; \mathit{sentp}(A, x, np) \;\wedge\; \mathit{sent}(A, x, np) > n\;) \\
\Rightarrow & \quad \{\ (7.12):\ \mathit{sentp}(A, x, np) \;\wedge\; \mathit{sent}(A, x, np) \leq A.\mathit{price}\ \ \} \\
 & A.\mathit{price} > n
\end{array}
$$

□

Lemma 7.2 (Increasing Price Lemma)

$$(\,\exists\, x \;::\; n < x.\mathit{MaxBid} \wedge \mathit{sentp}(A, x, np) \wedge \mathit{sent}(A, x, np) = n\;) \leadsto A.\mathit{price} > n$$

Proof:

$$
\begin{array}{ll}
 & (\,\exists\, x \;::\; n < x.\mathit{MaxBid} \;\wedge\; \mathit{sentp}(A, x, np) \;\wedge\; \mathit{sent}(A, x, np) = n\;) \\
\leadsto & \quad \{\ \text{channel property}\ \} \\
 & (\,\exists\, x \;::\; n < x.\mathit{MaxBid} \;\wedge\; \mathit{delp}(A, x, np) \;\wedge\; \mathit{del}(A, x, np) \geq n\;) \\
\Rightarrow & \\
 & (\,\exists\, x \;::\; (n < x.\mathit{MaxBid} \;\wedge\; \mathit{delp}(A, x, np) \;\wedge\; \mathit{del}(A, x, np) = n) \\
 & \;\vee\; (\mathit{delp}(A, x, np) \;\wedge\; \mathit{del}(A, x, np) > n)\;) \\
\Rightarrow & \quad \{\ \text{Lemma 7.1}\ \} \\
 & A.\mathit{price} > n \;\vee\; (\,\exists\, x \;::\; n < x.\mathit{MaxBid} \;\wedge\; \mathit{delp}(A, x, np) \;\wedge\; \mathit{del}(A, x, np) = n\;) \\
\leadsto & \quad \{\ (7.18)\ \} \\
 & A.\mathit{price} > n \;\vee\; (\,\exists\, x \;::\; (\mathit{delp}(A, x, np) \;\wedge\; \mathit{del}(A, x, np) > n) \\
 & \qquad\qquad\qquad\qquad \vee\; (\mathit{sentp}(x, A, \mathit{bid}) \;\wedge\; \mathit{sent}(x, A, \mathit{bid}) > n)\;) \\
\Rightarrow & \quad \{\ \text{Lemma 7.1}\ \} \\
 & A.\mathit{price} > n \;\vee\; (\,\exists\, x \;::\; \mathit{sentp}(x, A, \mathit{bid}) \;\wedge\; \mathit{sent}(x, A, \mathit{bid}) > n\;) \\
\leadsto & \quad \{\ \text{channel properties}\ \} \\
 & A.\mathit{price} > n \;\vee\; (\,\exists\, x \;::\; \mathit{delp}(x, A, \mathit{bid}) \;\wedge\; \mathit{del}(x, A, \mathit{bid}) > n\;)
\end{array}
$$

$$\Rightarrow \qquad \{\ (7.10)\text{:}\ \ (\forall x\ :\ delp(x, A, bid)\ :\ A.price \geq del(x, A, bid)\ )\ \ \}$$
$$A.price > n$$

□

Prove (7.1):

$$(\exists x\ ::\ A.Start < x.MaxBid\ ) \leadsto A.sold$$

Proof: Using the Increasing Price Lemma, we prove the following result:

$$A.price = n \leadsto A.price > n \vee (\forall x\ :\ x \neq A.winner\ :\ A.price \geq x.MaxBid\ )$$

$$A.price = n$$
$$\Rightarrow$$
$$(A.price = n\ \wedge\ (\exists x\ :\ x \neq A.winner\ :\ A.price < x.MaxBid\ ))$$
$$\vee\ (A.price = n\ \wedge\ (\forall x\ :\ x \neq A.winner\ :\ A.price \geq x.MaxBid\ ))$$
$$\Rightarrow \qquad \{\ (7.14)\text{:}\ \ (\forall x\ :\ x \neq A.winner\ :\ sentp(A, x, np)\ \wedge\ sent(A, x, np) = A.price\ )\ \ \}$$
$$(\exists x\ :\ x \neq A.winner\ :\ n < x.MaxBid\ \wedge\ sentp(A, x, np)\ \wedge\ sent(A, x, np) = n\ ))$$
$$\vee\ (A.price = n\ \wedge\ (\forall x\ :\ x \neq A.winner\ :\ A.price \geq x.MaxBid\ ))$$
$$\leadsto \qquad \{\ \text{Increasing Price Lemma 7.2}\ \ \}$$
$$A.price > n$$
$$\vee\ (A.price = n\ \wedge\ (\forall x\ :\ x \neq A.winner\ :\ A.price \geq x.MaxBid\ ))$$
$$\Rightarrow$$
$$A.price > n\ \vee\ (\forall x\ :\ x \neq A.winner\ :\ A.price \geq x.MaxBid\ )$$

From the result above, we can conclude by induction (since $A.price$ is an integer and all $MaxBid$ values are finite):

$$true \leadsto (\forall x\ :\ x \neq A.winner\ :\ A.price \geq x.MaxBid\ )$$

Now we conclude the proof:

$$(\forall x\ :\ x \neq A.winner\ :\ A.price \geq x.MaxBid\ )$$
$$\Rightarrow$$
$$(\forall x\ ::\ A.Start \geq x.MaxBid\ )$$
$$\vee\ ((\forall x\ :\ x \neq A.winner\ :\ A.price \geq x.MaxBid\ )\ \wedge\ (\exists x\ ::\ A.Start < x.MaxBid\ ))$$
$$\leadsto \qquad \{\ \text{Aside 7.1}\ \ \}$$
$$(\forall x\ ::\ A.Start \geq x.MaxBid\ )\ \vee\ A.sold$$

Since $Start$ and $MaxBid$ values are constants, this means we have the desired property:

$$(\exists x :: A.Start < x.MaxBid) \leadsto A.sold$$

This conclues the proof, with the exception of establishing the aside mentioned above.

□

Aside 7.1

$$(\forall x : x \neq A.winner : A.price \geq x.MaxBid)$$
$$\wedge (\exists x :: A.Start < x.MaxBid) \quad \leadsto \quad A.sold$$

Proof:

$$\begin{array}{ll}
 & (\forall x : x \neq A.winner : A.price \geq x.MaxBid) \wedge (\exists x :: A.Start < x.MaxBid) \\
\equiv & \quad \{ \text{ (7.14): } (\forall x : x \neq A.winner : sentp(A,x,np) \wedge sent(A,x,np) = A.price) \ \} \\
 & (\forall x : x \neq A.winner : sentp(A,x,np) \wedge sent(A,x,np) \geq x.MaxBid) \\
 & \wedge (\exists x :: A.Start < x.MaxBid) \\
\leadsto & \quad \{ \text{ channel properties and } \textbf{stable}.(A.Start < x.MaxBid) \ \} \\
 & (\exists w :: (\forall x : x \neq w : delp(A,x,np) \wedge del(A,x,np) \geq x.MaxBid)) \\
 & \wedge (\exists x :: A.Start < x.MaxBid) \\
\equiv & \quad \{ \text{ (7.15): } delp(A,x,np) \wedge del(A,x,np) \geq x.MaxBid \ \equiv \ sentp(x,A,nobid) \ \} \\
 & (\exists w :: (\forall x : x \neq w : sentp(x,A,nobid))) \wedge (\exists x :: A.Start < x.MaxBid) \\
\leadsto & \quad \{ \text{ channel properties and } \textbf{stable}.(A.Start < x.MaxBid) \ \} \\
 & (\exists w :: (\forall x : x \neq w : delp(x,A,nobid))) \wedge (\exists x :: A.Start < x.MaxBid) \\
\leadsto & \quad \{ \text{ Aside 7.2 and } \textbf{stable}.(delp(x,A,nobid)) \ \} \\
 & (\exists w :: (\forall x : x \neq w : delp(x,A,nobid))) \wedge A.price > A.Start \\
\equiv & \quad \{ \text{ Aside 7.3: } A.price > A.Start \Rightarrow \neg delp(A.winner,A,nobid) \ \} \\
 & (\exists w :: (\forall x : x \neq w : delp(x,A,nobid))) \wedge \neg delp(A.winner,A,nobid) \\
 & \wedge A.price > A.Start \\
\equiv & \\
 & (\forall x : x \neq A.winner : delp(x,A,nobid)) \wedge \neg delp(A.winner,A,nobid) \\
 & \wedge A.price > A.Start \\
\equiv & \quad \{ \text{ (7.11): } A.sold \ \equiv \ (\forall x : x \neq A.winner : delp(x,A,nobid)) \\
 & \qquad\qquad\qquad \wedge \neg delp(A.winner,A,nobid) \wedge A.price > A.Start \ \} \\
 & A.sold
\end{array}$$

□

Aside 7.2

$$(\exists x :: A.Start < x.MaxBid) \leadsto A.price > A.Start$$

Proof:

$$\begin{array}{ll}
 & (\exists x :: A.Start < x.MaxBid) \\
\Rightarrow & \quad \{ \text{ (7.7): } A.price \geq A.Start \ \} \\
 & A.price > A.Start \ \vee \ ((\exists x :: A.Start < x.MaxBid) \ \wedge \ A.price = A.Start) \\
\Rightarrow & \quad \{ \text{ (7.9): } A.price = A.Start \ \Rightarrow \ (\forall x :: sentp(A, x, np) \ \wedge \ sent(A, x, np) = A.price) \ \} \\
 & A.price > A.Start \\
 & \ \vee \ (\exists x :: A.Start < x.MaxBid \ \wedge \ sentp(A, x, np) \ \wedge \ sent(A, x, np) = A.Start) \\
\leadsto & \quad \{ \text{ Increasing Price Lemma 7.2 } \} \\
 & A.price > A.Start
\end{array}$$

□

Aside 7.3

$$A.price > A.Start \Rightarrow \neg delp(A.winner, A, nobid)$$

Proof:

$$\begin{array}{ll}
 & A.price > A.Start \\
\equiv & \quad \{ \text{ (7.8): } A.price > A.Start \ \Rightarrow \ delp(A.winner, A, bid) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \ del(A.winner, A, bid) = A.price \ \} \\
 & A.price > A.Start \ \wedge \ delp(A.winner, A, bid) \ \wedge \ del(A.winner, A, bid) = A.price \\
\Rightarrow & \quad \{ \text{ channel properties } \} \\
 & A.price > A.Start \ \wedge \ sentp(A.winner, A, bid) \ \wedge \ sent(A.winner, A, bid) \geq A.price \\
\Rightarrow & \quad \{ \text{ (7.17): } sentp(x, A, bid) \ \Rightarrow \ sent(x, A, bid) \leq x.MaxBid \ \} \\
 & A.price > A.Start \ \wedge \ A.price \leq A.winner.MaxBid \\
\Rightarrow & \quad \{ \text{ (7.8): } A.price > A.Start \ \Rightarrow \ sentp(A, A.winner, np) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \ sent(A, A.winner, np) < A.price \ \} \\
 & sentp(A, A.winner, np) \ \wedge \ sent(A, A.winner, np) < A.winner.MaxBid \\
\Rightarrow & \quad \{ \text{ channel properties } \} \\
 & \neg delp(A, A.winner, np) \ \vee \ del(A, A.winner, np) < A.winner.MaxBid \\
\equiv & \quad \{ \text{ (7.15): } delp(A, x, np) \ \wedge \ del(A, x, np) \geq x.MaxBid \ \equiv \ sentp(x, A, nobid) \ \} \\
 & \neg sentp(A.winner, A, nobid)
\end{array}$$

$$\Rightarrow \quad \{ \text{ channel properties } \}$$
$$\neg delp(A.winner, A, nobid)$$

□

Prove (7.2):

$$(\forall x :: A.Start \geq x.MaxBid) \leadsto (\forall x :: delp(x, A, nobid))$$

Proof: For any `Bidder` Component x ,

$$A.Start \geq x.MaxBid$$
$$\Rightarrow \quad \{ \text{ (7.13): } sentp(A, x, np) \wedge sent(A, x, np) \geq A.Start \ \}$$
$$sentp(A, x, np) \wedge sent(A, x, np) \geq x.MaxBid$$
$$\leadsto \quad \{ \text{ channel properties } \}$$
$$delp(A, x, np) \wedge del(A, x, np) \geq x.MaxBid$$
$$\equiv \quad \{ \text{ (7.15): } delp(A, x, np) \wedge del(A, x, np) \geq x.MaxBid \equiv sentp(x, A, nobid) \ \}$$
$$sentp(x, A, nobid)$$
$$\leadsto \quad \{ \text{ channel properties } \}$$
$$delp(x, A, nobid)$$

In conjunction with the stability of $delp(x, A, nobid)$, this gives the desired result.

□

Prove (7.3):

$$A.sold \Rightarrow (\forall x : x \neq A.winner : A.price \geq x.MaxBid)$$

Proof:

$$A.sold$$
$$\Rightarrow \quad \{ \text{ (7.11): } A.sold \Rightarrow (\forall x : x \neq A.winner : delp(x, A, nobid)) \ \}$$
$$(\forall x : x \neq A.winner : delp(x, A, nobid))$$
$$\Rightarrow \quad \{ \text{ channel properties } \}$$
$$(\forall x : x \neq A.winner : sentp(x, A, nobid))$$
$$\equiv \quad \{ \text{ (7.15): } delp(A, x, np) \wedge del(A, x, np) \geq x.MaxBid \equiv sentp(x, A, nobid) \ \}$$
$$(\forall x : x \neq A.winner : delp(A, x, np) \wedge del(A, x, np) \geq x.MaxBid)$$
$$\Rightarrow \quad \{ \text{ channel properties } \}$$

$$
\begin{array}{ll}
 & (\forall x \ : \ x \neq A.winner \ : \ sentp(A,x,np) \ \wedge \ sent(A,x,np) \geq x.MaxBid \ ) \\
\equiv & \qquad \{ \ (7.14)\text{:} \ \ (\forall x \ : \ x \neq A.winner \ : \ sentp(A,x,np) \ \wedge \ sent(A,x,np) = A.price \ ) \ \ \} \\
 & (\forall x \ : \ x \neq A.winner \ : \ A.price \geq x.MaxBid \ )
\end{array}
$$

□

Prove (7.4):

$$A.sold \ \Rightarrow \ A.winner.MaxBid \geq A.price$$

Proof:

$$
\begin{array}{ll}
 & A.sold \\
\Rightarrow & \qquad \{ \ (7.11)\text{:} \ \ A.sold \ \Rightarrow \ A.price > A.Start \ \ \} \\
 & A.price > A.Start \\
\Rightarrow & \qquad \{ \ (7.8)\text{:} \ \ A.price > A.Start \ \Rightarrow \ \ delp(A.winner, A, bid) \\
 & \qquad \qquad \qquad \qquad \qquad \qquad \wedge \ del(A.winner, A, bid) = A.price \ \ \} \\
 & delp(A.winner, A, bid) \ \wedge \ del(A.winner, A, bid) = A.price \\
\Rightarrow & \qquad \{ \ \text{channel properties} \ \ \} \\
 & sentp(A.winner, A, bid) \ \wedge \ sent(A.winner, A, bid) \geq A.price \\
\Rightarrow & \qquad \{ \ (7.17)\text{:} \ \ sentp(x, A, bid) \ \Rightarrow \ sent(x, A, bid) \leq x.MaxBid \ \ \} \\
 & A.winner.MaxBid \geq A.price
\end{array}
$$

□

Prove (7.5):

$$sentp(x,A,bid) \wedge sent(x,A,bid) \geq n \leadsto A.price \geq n$$

Proof:

$$
\begin{array}{ll}
 & sentp(x,A,bid) \ \wedge \ sent(x,A,bid) \geq n \\
\leadsto & \qquad \{ \ \text{channel properties} \ \ \} \\
 & delp(x,A,bid) \ \wedge \ del(x,A,bid) \geq n \\
\Rightarrow & \qquad \{ \ (7.10)\text{:} \ \ (\forall x \ : \ delp(x,A,bid) \ : \ A.price \geq del(x,A,bid) \ ) \ \ \} \\
 & A.price \geq n
\end{array}
$$

□

7.5 CORBA Instantiation of Solution

This system has been implemented in a commercially available CORBA-compliant distributed object framework, namely IBM's SOM/DSOM. The IDL descriptions of these objects are given here, in Programs 7.3 and 7.4. For clarity, the SOM/DSOM-specific elements of these interface declarations have been omitted here.

```
interface Auctioneer : SOMObject  {
   oneway void checkin (in Bidder b, in short id);
   oneway void bid (in short source, in long amount);
   oneway void nobid (in long amount);
};
```

Program 7.3: IDL definition of the `Auctioneer` interface.

```
interface Bidder : SOMObject  {
   oneway void newprice (in long price, in short id);
   oneway void congrats (in long price);
   oneway void winner (in long price, in short id);
};
```

Program 7.4: IDL definition of the `Bidder` interface.

The auctioneer implementation contains a method that was not present in the component definition, namely `checkin()`. This is a bootstrapping method used by the auctioneer to obtain references to bidders. The bidder implementation contains two extra methods, namely `congrats()` and `winner()`. These methods are used by the auctioneer to notify each bidder of the outcome of the auction. These methods are not material to the correctness of the application and so were excluded from our presentation and proof of the solution.

7.6 Discussion

We have proposed, specified, and implemented a solution to the distributed multiparty auction problem. The solution was rigorously proven to meet the required system specification. This proof took advantage of the simple compositional rule for certificates: A certificate that is a property of a component is also a property of any system containing that component.

The final step in system verification is the proof that the implementation of each component meets the specification of that component. One strength of certificates is that they are locally verifiable. Thus, each component can be reasoned about in isolation. Even the bidder's transience property is established strictly as a result of bidder actions and so can be unilaterally guaranteed by the bidder implementation.

An alternate approach to this verification is the validation (by testing) of component implementations against their specifications. As discussed in Chapter 5, certain certificates can be easily translated into run-time checks and warnings. We hypothesized that the subset of certificates to which this translation can be applied is a significant set, accounting for many of the certificates encountered in practice. Indeed, in the specifications of the `Auctioneer` and `Bidder` components we observe that all the certificates are either invariant, functional **next** , or functional **transient** properties. Notice that the **transient** property of the `Bidder` component is an example of a functional predicate with a disjunction. Both disjuncts imply the same value for the dummy variable, so the predicate is indeed functional.

It is interesting to note that the intuitive proof, in which an induction and boundedness argument is made on the price, is not the approach taken for the formal proof in Section 7.4. Such a strategy is possible for proving termination, but is complicated by the fact that the stable upper bound for the price cannot be calculated *a priori*. In particular, a price equal to the *second* highest maximum bid value may or may not be the final sale price, depending on the origin of the bid.

This final observation is an excellent example of the power of certificates. A simple description of object behavior can be captured by a few fundamental certificates, sufficient to make persuasive arguments of system properties. If a greater degree of confidence is desired, these certificates can be extended in an entirely consistent manner to form a more complete specification. The amount of rigor that is brought to bear can be tailored to the desired confidence in the correctness of the application.

Chapter 8

Example: A Branch and Bound Tree Search

In this chapter, we present another example that illustrates the utility of certificates in the development of a distributed application. An operational overview of the problem and solution is presented, followed by a formal specification of the system and of the individual components using certificates. A formal proof is given that the composition of these components meets the required system specification.

Unlike the previous example, some of the component certificates cannot be automatically translated into efficient run-time checks. This is because the definition of some of the bounds potentially involves prohibitively expensive calculation. Checks provided by the programmer, however, do allow the certificates to be tested at run-time. Also, the validity of the certificates can be formally established with an entirely local proof.

8.1 The Problem

We consider a distributed tree search based on the branch and bound algorithm [13, Chapter 3]. Each node of the tree being searched has an associated value. The goal of the search is to identify the leaf node with the maximum value.

The branch and bound tree search is based on the existence of two functions. The first function yields, for any node in the tree, a list of that node's children. This function describes how the tree branches, permitting the exploration of deeper levels. The second function returns, for any node in the tree, an upper bound on the values in the subtree of which that node is the root. A subtree can be removed from consideration if the upper bound associated with its root is less than or equal to the value of a known solution or a known lower bound of the solution. If both these functions can be calculated efficiently, this approach is a useful heuristic when the number of leaves is large.

The problem described in this chapter is a general one, as is the presentation of the solution. It is applicable to any branch and bound tree search. For illustration purposes, however, we choose a particular instance of this class of problems to implement in Section 2.3. The particular problem we solve is a generalization of the 0-1 knapsack problem [29]. We are given a list of items, each with an integer weight and value. We are also given a list of knapsacks, each with an integer capacity. The problem is to place items into knapsacks so as to maximize the total value carried in the knapsacks, subject to the constraint that none of the knapsack capacities are exceeded.

The tree to be explored is therefore the tree of partial solutions to this problem. A partial solution is an assignment of the first k items to knapsacks and a list of the unused capacities remaining for each knapsack. The root node of the tree contains no assignments of items to knapsacks and all knapsacks have their full capacity unused. A leaf of the tree contains a complete assignment of every item either to a knapsack or to not be included. The value of a node is the sum of the values of items that have been placed in knapsacks. The children of a node are the partial solutions which can be created by extending the parent node with the placement of the next item in the list. A bound on the subtree is determined by relaxing the constraint of the indivisibility of the items. The details of this calculation are not pertinent here. Indeed, the system will be specified and proven as a generic tree search algorithm, with references to the specific problem instance (*i.e.*, the generalized 0-1 knapsack problem) only where they are helpful for clarity.

8.2 A Solution

We implement here a solution based on the master-slave paradigm. A single master is responsible for dividing up the tree to be explored and assigning each section to a different slave (see Figure 8.1). The slaves perform the search in their respective partition and return the result to the master.

There are two sources of significant inefficiency in this approach. The first is the lack of load balancing. The partition assigned to one slave could be significantly more expensive to search than the others. The computation, however, cannot terminate until all slaves have reported back to the master. This can result in poor utilization of the slaves and suboptimal performance. Load balancing can be improved by a simple refinement. Instead

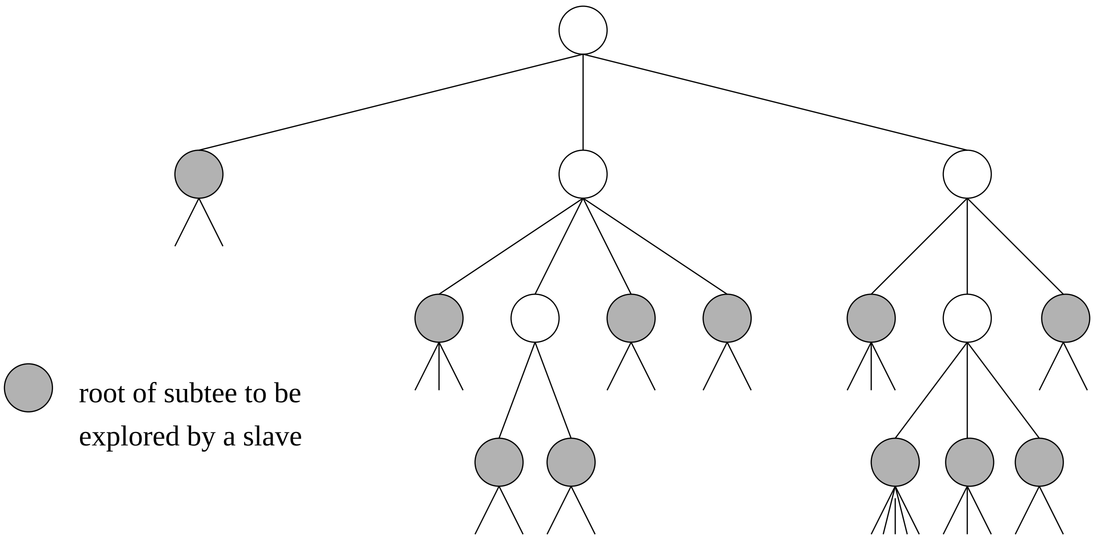

Figure 8.1: An example subdivision of a search tree by the master.

of returning the optimal value for an entire subtree, the slaves can return promising internal nodes to the master. The master then maintains a pool of work to be done (*i.e.*, the nodes that have not yet been explored) and idle slaves are given more work from this pool. This optimization complicates the presentation of the solution, but does not substantially modify the fundamental master-slave interaction. We therefore do not include this optimization here.

The second source of inefficiency in the approach outlined above is that slaves do not share bounds. That is, one slave could be searching a subtree whose value is not better than a lower bound already discovered by a different slave. To remedy this, slaves communicate the lower bounds they discover back to the master. The master is then responsible for broadcasting this information to the other slaves. This changes the master-slave interaction by creating a mixed node in the slave protocol. After a slave is given a subtree to explore, it may or may not receive lower bound updates. If none are received, the slave eventually completes the search and returns the optimal node found.

8.3 Master and Slave Components

Having described operationally the functionality of the master and slaves, we now present the component descriptions of these objects. The certificates for each component are given separately from the rest of the component description for readability.

8.3.1 Master

The master accepts two types of messages. The first encodes the solution discovered by the slaves and is given generically as `Tree_Node`. For the generalized knapsack problem this type is an assignment of some subset of items either to knapsacks or to be excluded from any knapsack. The second message type encodes lower bounds and is given generically as `Tree_Node_Value`. For the generalized knapsack problem this type is an integer representing the sum of values of items included in knapsacks. There is a single constant, `Prob`, representing the original problem to be solved (the root of the tree to be explored). In addition, there are two variables: `soln` (the optimal solution found by the slaves) and `done` (whether or not the computation has terminated). The master has three RPC targets: `lb`, `soln`, and `nosoln`. The first is used to report a lower bound. The second is used to return the optimal solution found by a slave. The third is used to signal that the subtree searched by the slave does not contain the optimal solution. Finally, as neighbors, the master maintains `pool`, a set of slave components. This component description is summarized in Program 8.1.

```
Component Master  {
  local types := Tree_Node, Tree_Node_Value
  local const := Prob : Tree_Node
  local vars :=  soln : Tree_Node
                 done : boolean
  rpc targets := lb (Tree_Node_Value) (monotonic)
                 soln (Tree_Node) (write-once)
                 nosoln (unary) (write-once)
  neighbors :=   pool : set of Component Slave
}
```

Program 8.1: Description of the `Master` component.

8.3.2 Slave

A slave accepts two types of messages: one (`Tree_Node`) encodes the subtree to be searched, and the other (`Tree_Node_Value`) encodes lower bounds discovered by other slaves. Each slave stores the subtree it is assigned in a local variable, `p`. This subtree is encoded by a single node, its root. A slave has two RPC targets, `prob` and `newlb`. The former is used to receive the subtree from the master, and the latter is used to receive updated lower bounds. Finally, each slave has a single neighbor, the master (`M`). This component description is summarized in Program 8.2.

```
Component Slave  {
  local types := Tree_Node, Tree_Node_Value
  local vars :=  p : Tree_Node
  rpc targets := prob (Tree_Node) (write-once)
                 newlb (Tree_Node_Value) (monotonic)
  neighbors :=   M : Component Master
}
```

Program 8.2: Description of the `Slave` component.

8.3.3 Certificates

In this section we give the certificate-based specification of the `Master` and `Slave` components introduced above.

We first introduce a function that allows us to abstract away from the details of the particular problem being solved. It permits us to treat the application as a generic search. The function is called opt_soln and for any given partial solution it returns the set of optimal solutions which can be formed by extending the partial solution. That is, for any node in the search tree, opt_soln gives the set of optimal (maximal) leaves in the subtree with that node as its root. For example, for the generalized knapsack problem this function gives all the assignments of items to knapsacks such that the total inside knapsacks is maximal. This function is a specification function and is not intended to be implemented directly.

We also introduce a function, $leaves$, that for any node in the search tree returns the set of leaves descended from that node. Thus, for a node P, we have:

$$leaves.P = (\cup i :: leaves.P_i) \quad \Rightarrow \quad (\exists i :: opt\_soln.P_i \subseteq opt\_soln.P)$$

Finally, we introduce a function, $value$, that for any leaf in the search tree returns the value of that leaf. Thus, for a problem P, we have:

$$l \in leaves.P \wedge value.l \geq opt\_val.P \quad \Rightarrow \quad l \in opt\_soln.P$$

The specification functions opt_val and opt_soln can now be expressed in terms of these two new functions. We use the convention of ordering solutions by their value. This allows a maximum to be taken over solutions. Thus, for a node P, we have:

$$\begin{aligned} opt\_val.P &= (\mathbf{Max}\, l : l \in leaves.P : value.l) \\ opt\_soln.P &= (\mathbf{Max}\, l : l \in leaves.P : P) \end{aligned}$$

Notice the difference in the two maximum operators above. In the first case the maximum is being taken over values of solutions, and returns a value, while in the second case it is being taken over solutions and returns the set of solutions with the highest value. We will continue to use these specification functions as convenient shorthands.

Master

The following certificates characterize the behavior of the `Master` component (where x is understood to range over `pool`, the set of available slaves):

- All slaves are sent a subproblem and the union of these subproblems covers all possible solutions to the original problem.

$$\textbf{invariant}.(\,(\forall x :: \ sentp(x, prob)\ )\\ \wedge\ (\cup x :: \ leaves.sent(x, prob)\ ) = leaves.Prob)$$

- A new lower bound is sent to a slave only if a slave has reported that value as a lower bound.

$$\textbf{invariant}.(sentp(x, newlb) \Rightarrow\\ (\exists y :: \ delp(y, lb) \wedge sent(x, newlb) \leq del(y, lb)\ ))$$

- The computation is complete exactly when all slaves have reported back to the master.

$$\textbf{invariant}.(done \equiv (\forall x :: \ delp(x, soln) \vee delp(x, nosoln)\ ))$$

- When the computation is done, the solution is set to the maximum-valued solution received from the slaves.

$$\textbf{invariant}.(done \Rightarrow soln \in (\,\textbf{Max}\, x : \ delp(x, soln) \ : \ del(x, soln)\ ))$$

Slave

The following certificates characterize the behavior of the `Slave` component:

- The slave reports back that no optimal solution exists in its subtree only if it has received a lower bound value that is better than the best solution in its subproblem.

$$\textbf{invariant}.(sentp(M, nosoln) \Rightarrow delp(M, prob) \wedge delp(M, newlb)\\ \wedge\ opt\_val.del(M, prob) < del(M, newlb))$$

- A lower bound is sent to the master only if a subproblem has been delivered and that lower bound is below the best solution in the subproblem given to this slave.

$$\mathbf{invariant}.(sentp(M, lb) \Rightarrow delp(M, prob) \\ \wedge sent(M, lb) \leq opt\_val.del(M, prob))$$

- A solution is sent to the master only if it is one of the optimal solutions for the subproblem assigned to this slave.

$$\mathbf{invariant}.(sentp(M, soln) \Rightarrow delp(M, prob) \\ \wedge sent(M, soln) \in opt\_soln.del(M, prob))$$

- The slave eventually returns either a solution or a report that the optimal solution is not in its subtree.

$$\mathbf{transient}.(delp(M, prob) \wedge \neg sentp(M, soln) \wedge \neg sentp(M, nosoln))$$

8.4 Proof of Correctness

8.4.1 Problem Specification

We can now state the specification for the branch and bound tree search. We use M to denote the single `Master` component in the system.

- The solution found is optimal. This safety condition states that when the calculation has terminated, the value stored in the master's local variable `soln` contains (one of) the optimal solution(s) to the original problem.

$$M.done \Rightarrow M.soln \in opt\_soln.(M.Prob) \tag{8.1}$$

- A solution is eventually found. This progress condition simply states that eventually the calculation does terminate.

$$true \leadsto M.done \tag{8.2}$$

8.4.2 Composition of Master and Slave Specifications

The composition of these components to form our distributed system consists of the binding of the neighbor values of the participating components. The set `pool` in the `Master` component is assigned the set of participating `Slave` components. Conversely, the neighbor value `M` of each slave is assigned the single `Master` component in the system.

Recall the simple compositional rule for certificates: A certificate of a component is also a certificate of any system of which that component is a part. Thus, the certificates given in Section 8.3.3 are also certificates for the entire system. We repeat these certificates here because, unlike their presentation in Section 8.3.3, the identification of channels is no longer clear from context, but must be made explicit. Also, as an abbreviation, we will omit the keyword **invariant** . Our convention is that properties that are not **next** , stable, **transient** , or leads-to, are understood to be invariant. The variable x is understood to range over the slaves in the system (*i.e.*, the elements of $M.pool$).

Master

$$(\forall x :: sentp(M, x, prob)) \wedge (\cup x :: leaves.sent(M, x, prob)) = leaves.(M.Prob) \tag{8.3}$$

$$sentp(M, x, newlb) \Rightarrow (\exists y :: delp(y, M, lb) \wedge sent(M, x, newlb) \leq del(y, M, lb)) \tag{8.4}$$

$$M.done \equiv (\forall x :: delp(x, M, soln) \vee delp(x, M, nosoln)) \tag{8.5}$$

$$M.done \Rightarrow M.soln \in (\mathbf{Max}\, x : delp(x, M, soln) : del(x, M, soln)) \tag{8.6}$$

From property (8.3) and the definition of the function $leaves$, we derive the following corollaries:

$$opt\_val.(M.Prob) = (\mathbf{Max}\, x :: opt\_val.sent(M, x, prob)) \tag{8.7}$$

$$leaves.sent(M, x, prob) \subseteq leaves.(M.Prob) \tag{8.8}$$

Slave

$$sentp(x, M, nosoln) \Rightarrow delp(M, x, prob) \wedge delp(M, x, newlb) \wedge opt\_val.del(M, x, prob) < del(M, x, newlb) \tag{8.9}$$

$$sentp(x, M, lb) \Rightarrow delp(M, x, prob) \wedge sent(x, M, lb) \leq opt\_val.del(M, x, prob) \tag{8.10}$$

$$sentp(x, M, soln) \Rightarrow delp(M, x, prob) \wedge sent(x, M, soln) \in opt\_soln.del(M, x, prob) \tag{8.11}$$

$$transient\ (delp(M, x, prob) \wedge \neg sentp(x, M, soln) \wedge \neg sentp(x, M, nosoln)) \tag{8.12}$$

From these properties (in particular (8.12)), we derive the following corollary:

$$delp(M, x, prob) \leadsto sentp(x, M, soln) \vee sentp(x, M, nosoln) \tag{8.13}$$

8.4.3 Proof of Solution

We begin with a lemma which is helpful in establishing the result.

Lemma 8.1

$$delp(x, M, soln) \Rightarrow del(x, M, soln) \in opt\_soln.sent(M, x, prob)$$

Proof:

$$\begin{array}{ll}
 & delp(x, M, soln) \\
\Rightarrow & \quad \{ \text{ channel properties (write-once) } \} \\
 & sentp(x, M, soln) \wedge del(x, M, soln) = sent(x, M, soln) \\
\Rightarrow & \quad \{ \text{ (8.11): } sentp(x, M, soln) \Rightarrow \\
 & \qquad\qquad delp(M, x, prob) \wedge sent(x, M, soln) \in opt_soln.del(M, x, prob) \ \} \\
 & delp(M, x, prob) \wedge del(x, M, soln) = sent(x, M, soln) \\
 & \ \wedge\ sent(x, M, soln) \in opt_soln.del(M, x, prob) \\
\Rightarrow & \quad \{ \text{ predicate calculus } \} \\
 & delp(M, x, prob) \wedge del(x, M, soln) \in opt_soln.del(M, x, prob) \\
\Rightarrow & \quad \{ \text{ channel property (write-once) } \} \\
 & sentp(M, x, prob) \wedge del(x, M, soln) \in opt_soln.sent(M, x, prob) \\
\Rightarrow & \quad \{ \text{ predicate calculus } \} \\
 & del(x, M, soln) \in opt_soln.sent(M, x, prob)
\end{array}$$

□

We are now ready to prove the result.

Prove (8.1):

$$M.done \Rightarrow M.Soln \in opt\_soln.(M.Prob)$$

Proof:

$$\begin{array}{ll}
 & M.done \\
\equiv & \quad \{ \text{ (8.5): } M.done \equiv (\forall x :: delp(x, M, soln) \vee delp(x, M, nosoln)\ ) \ \}
\end{array}$$

$$
\begin{array}{ll}
 & (\forall x :: \ delp(x, M, soln) \lor delp(x, M, nosoln)\ ) \\
\Rightarrow & \quad \{ \text{ channel properties } \} \\
 & (\forall x :: \ delp(x, M, soln) \lor sentp(x, M, nosoln)\ ) \\
\Rightarrow & \quad \{ \text{ (8.9): } sentp(x, M, nosoln) \Rightarrow \\
 & \qquad\qquad delp(M, x, prob) \land delp(M, x, newlb) \\
 & \qquad\qquad \land\ opt\_val.del(M, x, prob) < del(M, x, newlb) \ \} \\
 & (\forall x :: \ delp(x, M, soln) \lor (\ delp(M, x, prob) \land delp(M, x, newlb) \\
 & \qquad\qquad\qquad\qquad \land\ opt\_val.del(M, x, prob) < del(M, x, newlb))\ ) \\
\Rightarrow & \quad \{ \text{ channel property (write-once) } \} \\
 & (\forall x :: \ delp(x, M, soln) \lor (\ sentp(M, x, prob) \land delp(M, x, newlb) \\
 & \qquad\qquad\qquad\qquad \land\ opt\_val.sent(M, x, prob) < del(M, x, newlb))\ ) \\
\Rightarrow & \quad \{ \text{ predicate calculus } \} \\
 & (\forall x :: \ delp(x, M, soln) \lor (\ delp(M, x, newlb) \\
 & \qquad\qquad\qquad\qquad \land\ opt\_val.sent(M, x, prob) < del(M, x, newlb))\ ) \\
\Rightarrow & \quad \{ \text{ channel properties } \} \\
 & (\forall x :: \ delp(x, M, soln) \lor (\ sentp(M, x, newlb) \\
 & \qquad\qquad\qquad\qquad \land\ opt\_val.sent(M, x, prob) < sent(M, x, newlb))\ ) \\
\Rightarrow & \quad \{ \text{ (8.4): } sentp(M, x, newlb) \Rightarrow \\
 & \qquad\qquad (\exists y :: \ delp(y, M, lb) \land sent(M, x, newlb) \le del(y, M, lb)\ ) \ \} \\
 & (\forall x :: \ delp(x, M, soln) \\
 & \qquad\quad \lor (\ (\exists y :: \ delp(y, M, lb) \land sent(M, x, newlb) \le del(y, M, lb)\ ) \\
 & \qquad\qquad \land\ opt\_val.sent(M, x, prob) < sent(M, x, newlb))\ ) \\
\Rightarrow & \quad \{ \text{ predicate calculus } \} \\
 & (\forall x :: \ delp(x, M, soln) \\
 & \qquad\quad \lor\ (\exists y :: \ delp(y, M, lb) \land opt\_val.sent(M, x, prob) < del(y, M, lb)\ )\ ) \\
\Rightarrow & \quad \{ \text{ channel properties } \} \\
 & (\forall x :: \ delp(x, M, soln) \\
 & \qquad\quad \lor\ (\exists y :: \ sentp(y, M, lb) \land opt\_val.sent(M, x, prob) < sent(y, M, lb)\ )\ ) \\
\Rightarrow & \quad \{ \text{ (8.10): } sentp(x, M, lb) \Rightarrow \\
 & \qquad\qquad delp(M, x, prob) \land sent(x, M, lb) \le opt\_val.del(M, x, prob). \ \} \\
 & (\forall x :: \ delp(x, M, soln) \\
 & \qquad\quad \lor\ (\exists y :: \ delp(M, y, prob) \land sent(y, M, lb) \le opt\_val.del(M, y, prob) \\
 & \qquad\qquad\qquad \land\ opt\_val.sent(M, x, prob) < sent(y, M, lb)\ )\ ) \\
\Rightarrow & \quad \{ \text{ predicate calculus } \}
\end{array}
$$

$$
\begin{array}{ll}
& (\forall x :: \ delp(x, M, soln) \\
& \qquad \vee \ (\exists y :: \ delp(M, y, prob) \\
& \qquad\qquad \wedge \ opt\_val.sent(M, x, prob) < opt\_val.del(M, y, prob) \ ) \ ) \\
\Rightarrow & \quad \{ \text{ channel property (write-once) } \} \\
& (\forall x :: \ delp(x, M, soln) \\
& \qquad \vee \ (\exists y :: \ sentp(M, y, prob) \\
& \qquad\qquad \wedge \ opt\_val.sent(M, x, prob) < opt\_val.sent(M, y, prob) \ ) \ ) \\
\Rightarrow & \quad \{ \text{ predicate calculus } \} \\
& (\forall x :: \ delp(x, M, soln) \\
& \qquad \vee \ (\exists y :: \ opt\_val.sent(M, x, prob) < opt\_val.sent(M, y, prob) \ ) \ ) \\
\Rightarrow & \quad \{ \text{ calculus: } (\exists x :: \ \neg(\exists y :: \ opt\_val.sent(M, x, prob) \\
& \qquad\qquad\qquad < opt\_val.sent(M, y, prob) \ ) \ ) \quad \} \\
& (\exists x :: \ delp(x, M, soln) \\
& \qquad \wedge \ \neg(\exists y :: \ opt\_val.sent(M, x, prob) < opt\_val.sent(M, y, prob) \ ) \ ) \\
\equiv & \quad \{ \text{ calculus: definition of } \ \mathbf{max} \quad \} \\
& (\exists x :: \ delp(x, M, soln) \\
& \qquad \wedge \ opt\_val.sent(M, x, prob) = (\, \mathbf{Max}\, y :: \ opt\_val.sent(M, y, prob). \ ) \ ) \\
\equiv & \quad \{ \ (8.7){:} \ opt\_val.(M.Prob) = (\, \mathbf{Max}\, x :: \ opt\_val.sent(M, x, prob) \ ) \quad \} \\
& (\exists x :: \ delp(x, M, soln) \ \wedge \ opt\_val.sent(M, x, prob) = opt\_val.(M.Prob) \ ) \\
\equiv & \quad \{ \ \text{Lemma 8.1: } \ delp(x, M, soln) \ \Rightarrow \\
& \qquad\qquad del(x, M, soln) \in opt\_soln.sent(M, x, prob) \quad \} \\
& (\exists x :: \ delp(x, M, soln) \ \wedge \ del(x, M, soln) \in opt\_soln.sent(M, x, prob) \\
& \qquad \wedge \ opt\_val.sent(M, x, prob) = opt\_val.(M.Prob) \ ) \\
\Rightarrow & \quad \{ \ \text{calculus: } \ opt\_val.p_i = opt\_val.P \ \wedge \ leaves.p_i \subseteq leaves.P \\
& \qquad\qquad \Rightarrow \ opt\_soln.p_i \subseteq opt\_soln.P \quad \} \\
& (\exists x :: \ delp(x, M, soln) \ \wedge \ del(x, M, soln) \in opt\_soln.(M.Prob) \ ) \\
\Rightarrow & \quad \{ \text{ calculus: definition of } \ opt\_soln \quad \} \\
& (\exists x :: \ delp(x, M, soln) \ \wedge \ value.del(x, M, soln) = opt\_val.(M.Prob) \ ) \\
\Rightarrow & \quad \{ \text{ calculus: definition of } \ \mathbf{max} \quad \} \\
& (\, \mathbf{Max}\, x : \ delp(x, M, soln) : \ value.del(x, M, soln) \ ) \geq opt\_val.(M.Prob) \\
\equiv & \quad \{ \ \text{Lemma 8.1: } \ delp(x, M, soln) \ \Rightarrow \\
& \qquad\qquad del(x, M, soln) \in opt\_soln.sent(M, x, prob) \quad \} \\
& (\, \mathbf{Max}\, x : \ delp(x, M, soln) : \ value.del(x, M, soln) \ ) \geq opt\_val.(M.Prob) \\
& \wedge \ (\forall x : \ delp(x, M, soln) : \ del(x, M, soln) \in opt\_soln.sent(M, x, prob) \ )
\end{array}
$$

$$\equiv \quad \{ \text{ calculus: } opt\_soln.P \subseteq leaves.P \ \}$$
$$(\,\mathbf{Max}\, x \ : \ delp(x, M, soln) \ : \ value.del(x, M, soln)\ ) \geq opt\_val.(M.Prob)$$
$$\wedge\ (\forall\, x \ : \ delp(x, M, soln) \ : \ del(x, M, soln) \in leaves.sent(M, x, prob)\ )$$
$$\Rightarrow \quad \{ \text{ (8.8): } leaves.sent(M, x, prob) \subseteq leaves.(M.Prob) \ \}$$
$$(\,\mathbf{Max}\, x \ : \ delp(x, M, soln) \ : \ value.del(x, M, soln)\ ) \geq opt\_val.(M.Prob)$$
$$\wedge\ (\forall\, x \ : \ delp(x, M, soln) \ : \ del(x, M, soln) \in leaves.(M.Prob)\ )$$
$$\Rightarrow \quad \{ \text{ calculus: definition of } \mathbf{max} \ \}$$
$$(\,\mathbf{Max}\, x \ : \ delp(x, M, soln) \ : \ value.del(x, M, soln)\ ) \geq opt\_val.(M.Prob)$$
$$\wedge\ (\,\mathbf{Max}\, x \ : \ delp(x, M, soln) \ : \ del(x, M, soln)\ ) \subseteq leaves.(M.Prob)$$
$$\Rightarrow \quad \{ \text{ calculus: } l \in leaves.P \ \wedge\ value.l \geq opt\_val.P \ \Rightarrow\ l \in opt\_soln.P \ \}$$
$$(\,\mathbf{Max}\, x \ : \ delp(x, M, soln) \ : \ del(x, M, soln)\ ) \subseteq opt\_soln.(M.Prob)$$

So we conclude the proof:

$$M.done$$
$$\Rightarrow \quad \{ \text{ (8.6): } M.done \ \Rightarrow \ M.soln \in (\,\mathbf{Max}\, x \ : \ delp(x, M, soln) \ : \ del(x, M, soln)\ ) \ \}$$
$$M.done \ \wedge\ M.soln \in (\,\mathbf{Max}\, x \ : \ delp(x, M, soln) \ : \ del(x, M, soln)\ )$$
$$\Rightarrow \quad \{ \text{ above: } M.done \ \Rightarrow\ (\,\mathbf{Max}\, x \ : \ delp(x, M, soln) \ : \ del(x, M, soln)\ ) \subseteq opt\_soln.(M.Prob) \ \}$$
$$M.soln \in (\,\mathbf{Max}\, x \ : \ delp(x, M, soln) \ : \ del(x, M, soln)\ )$$
$$\wedge\ (\,\mathbf{Max}\, x \ : \ delp(x, M, soln) \ : \ del(x, M, soln)\ ) \subseteq opt\_soln.(M.Prob)$$
$$\Rightarrow \quad \{ \text{ calculus } \}$$
$$M.soln \in opt\_soln.(M.Prob)$$

$\square$

Prove (8.2):

$$true \leadsto M.done$$

Proof:

$$true$$
$$\equiv \quad \{ \text{ (8.3): } (\forall\, x \ :: \ sentp(M, x, prob)\ ) \ \wedge\ (\cup\, x \ :: \ leaves.sent(M, x, prob)\ ) = leaves.(M.Prob) \ \}$$
$$(\forall\, x \ :: \ sentp(M, x, prob)\ ) \ \wedge\ (\cup\, x \ :: \ leaves.sent(M, x, prob)\ ) = leaves.(M.Prob)$$
$$\Rightarrow$$

$$
\begin{array}{ll}
 & (\forall x :: sentp(M, x, prob)) \\
\leadsto & \quad \{ \text{ channel properties } \} \\
 & (\forall x :: delp(M, x, prob)) \\
\leadsto & \quad \{ \text{ (8.13): } delp(M, x, prob) \leadsto sentp(x, M, soln) \vee sentp(x, M, nosoln) \} \\
 & (\forall x :: sentp(x, M, soln) \vee sentp(x, M, nosoln)) \\
\leadsto & \quad \{ \text{ channel properties, twice } \} \\
 & (\forall x :: delp(x, M, soln) \vee delp(x, M, nosoln)) \\
\equiv & \quad \{ \text{ (8.5): } M.done \equiv (\forall x :: delp(x, M, soln) \vee delp(x, M, nosoln)) \} \\
 & M.done
\end{array}
$$

□

8.5 CORBA Instantiation of Solution

This system has been implemented in a commercially available CORBA-compliant distributed object framework, namely IBM's SOM/DSOM. The IDL descriptions of these objects are given here, in Programs 8.3 and 8.4. For clarity, the SOM/DSOM-specific elements of these interface declarations have been omitted here.

```
interface Master : SOMObject  {
   oneway void lb (in long amount);
   oneway void soln (in long amount);
   oneway void nosoln ();
};
```

Program 8.3: IDL definition of the `Master` interface.

```
interface Slave : SOMObject  {
   oneway void prob (in ProblemType problem);
   oneway void newlb (in long amount);
};
```

Program 8.4: IDL definition of the `Slave` interface.

8.6 Discussion

We have proposed and specified a system that performs a generic tree search in a branch-and-bound manner. The solution was rigorously proven to meet the required system specification. This proof took advantage of the simple compositional rule for certificates: A certificate that is a property of a component is also a property of any system containing that component. We have also implemented the solution to a specific instance of such a search: the generalized 0-1 knapsack problem.

We observe that the certificates used in the specification of the `Master` and `Slave` components are all simple certificates. There is a single **transient** property (in the specification of `Slave`) and it is an unquantified **transient** property. We can therefore hope that, as with the distributed auction example in Chapter 7, much of the testing harness for this application can be generated automatically.

Some of the certificates given in this example differ in an important way from those of the distributed auction example. In particular, (8.9), (8.10), and (8.11) all make use of one of the specification functions *opt_val* or *opt_soln*. Although the value of these functions is uniquely determined by the state of the slave (in particular, the tree node the slave received from the master), it is extremely expensive to calculate. Indeed, this calculation is precisely the task assigned to the slave. It is therefore unreasonable to expect that an efficient run-time check of these certificates could be automatically generated.

Nevertheless, it is still possible for the programmer to implement functions that approximate the boundedness expressed by these certificates. For example, (8.10) states that any lower bound sent by the slave is bounded above by the value of the optimal solution of the subtree being searched. To check that this is the case, a function can be written that returns the value of some promising leaf node in the subtree. In fact, this is the function written by the implementor of the slave component to find a lower bound. A run-time check that the leaf node returned by this function is indeed a descendant of the original tree node received by the slave is therefore sufficient to test the certificate. Clearly such a test requires the insight of a human programmer.

On the surface, there appear to be several similarities between the tree search application presented here and the distributed auction presented in Chapter 7. Both have a single central server and a star topology of interconnections. The bids submitted in the auction are analogous to the lower bounds, and eventually solutions, submitted in the tree search. The process of selecting a sale price is analogous to finding the leaf with the greatest value. It is curious, therefore, that the proof of the branch-and-bound tree search system is much shorter than the proof of the distributed auction.

This brevity is due to an important difference: the difference in the termination condition. In the distributed auction, the auctioneer cannot know whether a bidder is willing to bid more, or whether the bid is equal to that bidder's maximum. Terminating an auction, then, requires there to be at most one active bidder and the rest to have declined further participation. In the branch-and-bound tree search, however, the final solution (or bid) is distinguished from the lower bounds. It is sufficient that all slaves have replied in some manner, either with a solution or with an indication that their subtree is not relevant to the solution. In either case, this final "bid" is distinctive and the computation can terminate when all slaves have replied. Viewing the branch-and-bound tree search as an auction, a lower bound reported by a slave is analogous to a promise to bid at least that amount. A solution reported by a slave is a bid. Although such an auction has a simpler proof than the one given in Chapter 7, it requires the winning bidder to submit its maximum bid value. As discussed in Chapter 7, such a solution is not satisfactory.

Chapter 9

Related Work

In this chapter we outline some related areas of research. The relevant results from these areas are compared with our own approach.

9.1 Specification Theory

Writing correct software means writing software that meets a specification. The problem of writing correct specifications, of course, remains. However, specifications need not be executable by a computer, which means they can be easier to write than programs for several reasons: (i) they can use high-level intuitive abstractions that may not be available in the implementation language, and (ii) they can contain less information than a program since they focus on the problem to be solved rather than on the method of solution. Several approaches to the specification of concurrent and distributed systems have emerged.

9.1.1 Axiomatic

Axiomatic specifications define fundamental language constructs by axioms that can then be composed using rules of inference to form more complicated language constructs. This approach has been attributed to Floyd [26] and has been applied with success to sequential systems. The two primary forms of this style of specification for sequential programs are Hoare triples [40] and Dijkstra's weakest precondition [24, 25].

Axiomatic specification has also been demonstrated to be a powerful mechanism in the context of concurrent systems. Martin's seminal paper [63] gave an axiomatic definition of synchronization primitives (such as send-receive) in terms of boundedness, progress, and fairness. Attempts have also been made to extend the approaches taken with sequential systems to concurrent ones. For example, Hoare triples were extended by Owicki and Gries [79, 78] with a requirement to establish noninterference between threads of execution. An extension of Dijkstra's weakest precondition was the notion of weakest and strongest invariants (*win* and *sin*) [54].

An alternate approach to the specification of components of concurrent systems has been the definition of component behavior given that its environment has a certain behavior. That is, each component in a system must behave correctly only if the other components do. The circularity of this reasoning has been broken by various proposals, including rely-guarantee [46], hypothesis-conclusion [19], assumption-commitment [22], offers-using [52], and assumption-guarantee [1]. For example, the assumption-guarantee approach of Abadi and Lamport restricts the assumptions of environment behavior to safety properties. The modified rely-guarantee approach of Manohar and Sivilotti [62] allows progress properties to be part of the rely clause, but requires the explicit construction of an acyclic implication ladder. The weakest guarantee approach of Chandy and Sanders [20, 21] breaks the circularity by considering requirements on the entire system rather than just the environment. This approach characterizes two kinds of component properties: exists-component (if any component of a system has such a property, the entire system has the property), and all-component (if all components of a system have such a property, the entire system has the property). Our certificates are examples of "exists-component" properties.

9.1.2 Temporal Logic

Modal logic, which has been studied since its appearance as a syllogism in Aristotle's *De interpretatione*, adds the operators "necessarily" and "possibly" to the usual ones of propositional logic (*e.g.*, $\wedge$, $\vee$, $\neg$, $\Rightarrow$) [44]. Temporal logic can be viewed as a branch of modal logic in which these operators are given a temporal interpretation [83, 84]. The use of temporal logic to formalize the behavior of computer systems was first proposed by Kröger for sequential systems [50], and by Pnueli for concurrent ones [82]. A computation is viewed as a sequence of global states, and properties of a system are then given as temporal properties of these sequences. Different characterizations of the sets of sequences to which the temporal operators apply lead to different versions of temporal logic. For example, the structure of the sequence may be linear or branching, finite or infinite. A common formulation, based on linear temporal logic, is sometimes called Manna-Pnueli theory [61, 51].

Several specification notations with strong ties to temporal logic have been developed. The UNITY model [19] defines a program to be a collection of assignment statements from which, repeatedly, a statement is chosen for execution. The model gives a fairness requirement for this selection. Program properties are then given in terms of *unless*, *ensures*, and *leads-to*. TLA [56] (standing for "temporal logic of actions") is a logic for system specification based on the fundamental operators $\Box$ ("always") and $\Diamond$ ("eventually") of temporal logic. TLA allows for the expression of different fairness requirements (known as strong and weak), as well as the hiding of variables in specifications. Both UNITY and TLA allow for stuttering (*i.e.*, the repetition of global state in the trace of a computation). Implementation-language-specific notations that incorporate temporal logic have also been proposed, such as COL [48]. This specification language and its associated deadlock analyzer are part of an Ada [2] design environment. In [36], a procedural monitor-based programming language, VALET, is defined with a temporal logic semantics.

Our fundamental operators (**initially** , **next** , and **transient**) are based on well-known operators in temporal logic. In particular, [69, 68] contains an excellent exposition of the latter two. Our derived operators (**stable** , **invariant** , and leads-to) are also familiar from many temporal frameworks, but were inspired by their presentation in UNITY. The **follows** operator is similar to the *detects* operator in UNITY in that it combines safety and progress, and is similar to the method used in [18] to reason about continuous systems. Unlike *detects*, the **follows** operator deals with monotonic variables. This permits the derivation of several useful properties (*e.g.*, two fixed point theorems, a function application theorem, as well as juncitivity, union, and intersection theorems) that *detects* does not enjoy.

9.1.3 Calculational Refinement

An implementation can be viewed as a specification that happens to be executable by a computer. That is, specifications and implementations are distinguished only by implementability. This view has given rise to the study of the refinement from specification to implementation in a calculational manner. The refinement calculus, as it is known, was pioneered in [4], [73], and [72]. A calculation approach to specification refinement has also met with considerable success in the area of asynchronous VLSI design. A CSP-like [41] specification is refined through a series of program transformations into a hardware implementation [65, 64].

Component specifications based on our certificates do form a lattice that is partially ordered by refinement. The ordering relationship is given by implication and can be established using propositional calculus as illustrated in Chapters 7 and 8. Providing a refinement path from specification to implementation is not, however, the focus of our approach.

9.2 Specification Languages and Notations

Several specification notations have found use in the development of real systems. Larch [35, 58] is a family of languages that support a two-tiered style of specification. One tier is written in a common notation (the Larch Shared Language), while the other is written in an interface language tailored for a particular programming notation. Interface languages have been designed for C [35], Modula-3 [35], Ada [34], and C++ [58] among others. Another popular specification language, SDL [90], is used for the description of telecommunication systems, as is the Estelle language [14]. Both these notations are based on an extended finite state machine abstraction. VDM (the Vienna Development Method) [8, 47] has also enjoyed success in its application to large systems such as the specification of PL/I and Ada [9].

These specification languages can all be used to guide the selection of test data. They are also consistent with the automatic generation of testing harnesses for implementations.

9.3 Software Validation

Validation is the process whereby software is tested to ensure that it complies with its specification [45]. Of course, such testing cannot guarantee the absence of errors in a real system [42], but it can increase programmer confidence in the correctness of the system. Much work has been done on the development of test selection strategies, beginning with Goodenough and Gerhart's seminal paper [31]. There are two fundamental approaches to the selection of test data: (i) selecting tests based on the system specification, and (ii) selecting tests based on the structure of the implementation. Both approaches are discussed and further classified in [71].

TSL (Task Sequencing Language) is a language for formally specifying the behavior of concurrent Ada programs, by specifying sequences of tasking events that can occur or are explicitly not allowed [38]. Work has been done towards the translation of these specifications into run-time checks [39]. This work has continued in the form of Rapide, an executable architecture definition language [60].

Another specification language designed with automated testing in mind is ADL by Sun Microsystems and the X/Open organization [89]. Originally built for the specification of C functions, it has been extended to C++ and Java [98]. In conjunction with a test-data description file, an ADL description is used to automatically generate a test program. Functional specifications are given as method postconditions, however, so they have a strong client-server bias. ADL is therefore less appropriate for the specification of peer-to-peer style distributed computations.

9.4 CORBA IDL Extensions

Some of the notations discussed above have also been proposed as extensions to CORBA IDL. For example, a Larch interface language for use with CORBA IDL has been studied [92]. ADL has also been explored as an extension of CORBA IDL [88]. Both approaches base their functional specification on preconditions and postconditions of object methods. CDL (Constraint Design Language) is a language supported as an IDL extension and used for expressing constraints on object declarations [49]. This notation, however, is intended to support the definition of architectural elements rather than the functional specification of components.

9.5 Component Technology

The modular development of programs has long been recognized as critical for the practical development of large systems [80]. At the same time, the high cost of construction of reliable software systems has made reuse (of implementation, architecture, and reasoning) a priority in many organizations. The confluence of these two forces makes component-based technology attractive in a competitive software market. One of the proponents of commercial off-the-shelf (COTS) component assembly for software systems is the Software Engineering Institute at Carnegie Mellon University (and, in particular, the CBS initiative underway there) [15].

One of the challenges to COTS component assembly is establishing confidence in the correctness of the component being used. The Fox project supports the concept of "proof-carrying code" [76]. That is, a proof of correctness is embedded in the assembled implementation code, which the user can verify. This approach protects both the user of the component (from incorrect implementations) and the developer of the component (maintaining the privacy of the implementation code). It is limited, however, to safety properties that are appropriate for verification by a type-checking algorithm. The confidence problem has also been addressed by extending component implementations with result-checking code [11]. The central idea is that it can be easier (*e.g.*, faster or more space efficient) to check the result of a computation than to perform the computation itself [10]. These simple checkers have been proposed and examined in a debugging role, but the extension of such checkers to a security role is an intriguing consideration.

Chapter 10

Conclusion

10.1 Summary

We have presented a specification methodology that addresses three parts of the development cycle for distributed object systems: (i) the specification of systems and components, (ii) the compositional reasoning used to verify that a collection of components satisfy a system specification, and (iii) the validation of component implementations. For the specification of system and component behavior, we use a collection of temporal operators, based on three fundamental operators: **initially** , **next** , and **transient** . From these, we derived some familiar operators, **invariant** , **stable** , and leads-to, as well as a less familiar (but equally useful, as we have found) operator, **follows** . What characterizes our methodology is the use of these operators in a restricted manner:

- A specification statement can refer only to the state of a single component.
- A single component must be able to guarantee the validity of a specification statement regardless of the environment in which it is placed.

The **follows** operator discussed in Chapter 4 has several pleasing properties, as we have demonstrated through its use in a succinct proof of the earliest meeting time problem. The operator is restricted to monotonic variables, but this monotonicity can be defined over an arbitrary lattice. Thus, this specialized operator can find general applicability in distributed systems, at least to the extent that such monotonic variables occur. For example, channels, whether ordered or unordered, can be viewed as monotonic variables (*i.e.*, either sequences or sets of messages). With this interpretation, **follows** was invaluable in the proof of channel properties in Chapter 4. Similar proofs of these same properties without using **follows** occupy 25 pages and are, in our opinion, less readable.

We have characterized a subset of our general certificates that can be mapped automatically into a testing harness embedded in the implementation stub code. This subset is general enough to include the most common uses of certificates, such as specifying monotonicity and boundedness. This approach is consistent with CORBA IDL philosophy of generating stub code from class declarations. Of course, our methodology is not restricted to the CORBA framework. However, CORBA did provide a natural context for our work, as our certificates can be seen as IDL extensions, and the generation of the testing harness can be easily integrated with the CORBA development cycle. For certificates in the characterized subset, we have defined the mapping of these certificates into an implementation language (*viz.*, C++).

Services help alleviate some of the difficulties of proving properties of systems. For example, consider a system in which each component is eventually assigned a stable, unique integer. This behavior could be proven from the protocol expressed by the individual components' certificates. However, this approach requires that the result be reproven for any combination of components that have this simple behavior as part of their specification. Instead, a service that implements this system behavior can be defined and proven. This service can then be reused without repetition of the proof. In this thesis, we have provided two services: tokens and logical clocks. There is anecdotal evidence that these two services span a considerable range of applications. The degree to which services can alleviate the difficulty of proof reuse can be quantified only by a large scale deployment of our methodology, which is outside the scope of this thesis. We are encouraged by our work with tokens and clocks, however, and believe that, in practice, a relatively small number of services account for a large number of applications with non-conjunctive properties.

Our experiments — the application of our methodology to a distributed auction and to a distributed tree search algorithm — convince us of the utility of our approach. We envision an active marketplace of off-the-shelf components, dynamically assembled by consumers into distributed systems. The methodology presented in this thesis provides the groundwork for the viability of such a marketplace.

10.2 Future Work

There are several natural extensions to the work presented here. These extensions build on our methodology and are all consistent with our fundamental goal of supporting, in a practical way, the creation of correct distributed systems.

First, the tool described in Chapter 5 could be implemented. This chapter describes a subset of certificates that are appropriate for automatic translation into a testing harness. All implementations of the CORBA standard include an IDL parser that takes as input the class declarations and produces as output the skeleton for the implementation code. Such a parser could be modified to accept our component descriptions as extensions to the usual IDL declaration and produce an implementation skeleton that is augmented with the testing harness according to the mappings given in Chapter 5.

Another intriguing possibility is the use of certificates as a run-time check by component consumers, rather than component developers. A component implementation could be published or sold as an executable program, a certificate-based specification, and an externally verifiable checker. The proprietary code of the vendor would be protected since the component implementation would not be directly available to the consumer. On the other hand, the checker code would be available to the consumer, who would then be able to verify that this checker does indeed test the maintenance of the component properties promised by the certificates. The security of the consumer would be protected by allowing the component to be run in a safe mode, where interactions between the component and the rest of the system would be monitored by the verified checker.

Although our certificates enjoy a simple compositional rule, formal proofs of distributed systems require some effort and sophistication. It is therefore desirable to reuse these proofs as much as possible. For certain fundamental collections of certificates, this was achieved by the introduction of services. Here, however, we have not addressed the reuse of proofs for general systems. If component A is a refinement of component B, A can always be used in place of B and the system will behave correctly. A component provider should therefore be able to reuse the proofs involving any component being refined. There is no conceptual difficulty with this reuse: Refinement is defined by specification implication and proof reuse follows as a mathematical consequence. The challenge is to provide an infrastructure to help component implementors locate publicly available proofs that are appropriate for them to use. The initial steps towards such an infrastructure are outlined in [94], where we describe how a universal distributed type hierarchy could be established and maintained. Such a system could also address the issue of the dynamic location of components based on semantic information, in addition to the syntactic information (*e.g.*, component name) used in today's technologies.

Bibliography

[1] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[2] American National Standards Institute, Inc. *The Programming Language Ada Reference Manual.* Springer-Verlag, Berlin, Germany, 1983. ANSI/MIL-STD-1815A.

[3] Krzysztof R. Apt. Correctness proofs of distributed termination algorithms. *ACM Transactions on Programming Languages and Systems*, 8(3):388–405, July 1986.

[4] R. J. R. Back. On correct refinements of programs. *Journal of Computer and System Sciences*, 23:49–68, 1981.

[5] T. J. Berners-Lee, R. Cailliau, J-F Groff, and B. Pollermann. World-wide web: The information universe. *Electronic Networking: Research, Applications, and Policy*, 1(2):52–58, Spring 1992.

[6] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The world-wide web. *Communications of the ACM*, 37(8):76–82, August 1994.

[7] Garrett Birkhoff. *Lattice Theory.* American Mathematical Society, New York, New York, 1940.

[8] Dines Bjørner and Cliff B. Jones. *Formal Specification and Software Development.* Prentice/Hall, Englewood Cliffs, New Jersey, 1982.

[9] Dines Bjørner and O. N. Oest. *Towards a Formal Description of Ada.* Number 98 in Lecture Notes in Computer Science. Springer-Verlag, New York, New York, 1980.

[10] Manuel Blum. Designing programs to check their work. Technical Report TR-88-009, International Computer Science Institute, 1988.

[11] Manuel Blum and Hal Wasserman. Program result-checking: a theory of testing meets a test of theory. In *Proceedings of the 35th Symposium on the Foundations of Computer Science*, pages 382–392, 1994.

[12] Grady Booch. *Object-Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, California 94065, 1991.

[13] Gilles Brassard and Paul Bratley. *Algorithmics: Theory and Practice.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1988.

[14] S. Budkowski and P. Dembinski. An introduction to Estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1):3–23, 1987.

[15] David Carney. Assembling large systems from COTS components: Opportunities, cautions, and complexities. SEI monograph, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1997.

[16] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming language. Technical Report CS-TR-92-01, Computer Science Department, California Institute of Technology, 256-80 Caltech, Pasadena, California 91125, 1992.

[17] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[18] K. Mani Chandy. Reasoning about continuous systems. *Science of Computer Programming*, 14(2–3):117–132, October 1990.

[19] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[20] K. Mani Chandy and Beverly A. Sanders. Predicate transformers for reasoning about concurrent computation. *Science of Computer Programming*, 24(2):129–148, April 1995.

[21] K. Mani Chandy and Beverly A. Sanders. Predicate transformers for reasoning about concurrent computation (volume 24, page 129, 1995). *Science of Computer Programming*, 29(3):335, September 1997. Correction.

[22] Pierre Collette. Composition of assumption-commitment specification in a UNITY style. *Science of Computer Programming*, 23:107–125, 1994.

[23] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, New York, 1968.

[24] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hass Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[25] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, 175 Fifth Avenue, New York, New York 10010, 1990.

[26] Robert W. Floyd. Assigning meanings to programs. In *Proceedings Symposium on Applied Mathematics*, volume 19, pages 19–31, 1967.

[27] Ian T. Foster and K. Mani Chandy. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 25(1), February 1995.

[28] Nissim Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55, January 1980.

[29] M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness.* Freeman, San Francisco, California, 1983.

[30] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 55 Hayward Street, Cambridge, Massachusetts 02142-1399, 1994.

[31] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, June 1975.

[32] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[33] M. Gouda, E. Manning, and Y. T. Yu. On the progress of communication between two finite state machines. *Information and Control*, 63:200–216, April 1983.

[34] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, September 1990.

[35] John V. Guttag, James J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, New York, 1993.

[36] B. Hailpern. *Verifying Concurrent Processes Using Temporal Logic.* Number 129 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1982.

[37] Eric C. R. Hehner. A practical theory of programming. *Science of Computer Programming*, 14(2–3):133–158, October 1990.

[38] D. P. Helmbold and D. C. Luckham. TSL: Task sequencing language. In *Ada in Use: Proceedings of the Ada International Conference*, pages 255–274. Cambridge University Press, May 1985.

[39] David P. Helmbold and Douglas L. Bryan. Design of run time monitors for concurrent programs. Technical Report CSL-TR-89-395, Computer Systems Laboratory, Stanford University, October 1989.

[40] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[41] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[42] W. E. Howden. *Functional Program Testing and Analysis.* McGraw-Hill, New York, New York, 1987.

[43] D. A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257:3–4, 161–190, and 275–303, 1954.

[44] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic.* Methuen, New York, New York, 1968.

[45] IEEE. *Standard Glossary of Software Engineering Terminology.* IEEE, February 1991. ANSI/IEEE Std 610.12-1990.

[46] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[47] Cliff B. Jones. *Systematic Software Development Using VDM.* Series in Computer Science. Prentice-Hall International, Englewood Cliffs, New Jersey 07632, 1986.

[48] G. M. Karam and R. J. A. Buhr. Temporal logic-based deadlock analysis for Ada. *IEEE Transactions on Software Engineering*, 17(10):1109–1125, 1991.

[49] Nils Klarlund, Jari Koistinen, and Michael I. Schwartzbach. Formal design constraints. *ACM SIGPLAN Notices*, 31(10):370–383, October 1996.

[50] Fred Kröger. LAR: A logic of alogrithmic reasoning. *Acta Informatica*, 8:243–266, 1977.

[51] Fred Kröger. *Temporal Logic of Programs*, volume 8 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1987.

[52] S. S. Lam and A. U. Shankar. A theory of interfaces and modules 1: Composition theorem. *IEEE Transactions on Software Engineering*, 20(1):55–71, January 1994.

[53] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[54] Leslie Lamport. win and sin: Predicate transformers for concurrency. Technical Report 17, Digital Systems Research Center, Palo Alto, California, May 1987. Revised December 1989.

[55] Leslie Lamport. A theorem on atomicity in distributed algorithms. *Distributed Computing*, 4(2):59–68, 1990.

[56] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[57] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? Technical Report 147, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, May 1997.

[58] Gary T. Leavens. An overview of Larch/C++: Behavioural specifications for C++ modules. Technical Report TR #96-01c, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011-1040, February 1996. Revised January 1997.

[59] A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. Gordon and Breach, Science Publishers, Inc., 150 Fifth Avenue, New York, New York 10011, 1969.

[60] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995. Special Issue on Software Architecture.

[61] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume 1. Specification. Springer-Verlag, 175 Fifth Avenue, New York, New York 10010, 1992.

[62] Rajit Manohar and Paolo A. G. Sivilotti. Composing processes using modified rely-guarantee specifications. Technical Report CS-TR-96-22, Computer Science Department, California Institute of Technology, 256-80 Caltech, Pasadena, California 91125, June 1996.

[63] Alain J. Martin. An axiomatic definition of synchronization primitives. *Acta Informatica*, 16:219–235, 1981.

[64] Alain J. Martin. *Formal methods for VLSI design*, chapter Synthesis of Asynchronous VLSI Circuits. Elsevier Science Publishing Company, New York, New York, 1990.

[65] Alain J. Martin. Synthesis of asynchronous VLSI circuits. Technical Report Caltech-CS-TR-93-28, Computer Science Department, California Institute of Technology, 256-80 Caltech, Pasadena, California 91125, 1993.

[66] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[67] Bertrand Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice Hall, 1988.

[68] Jayadev Misra. A logic for concurrent programming: Progress. *Journal of Computer & Software Engineering*, 3(2):273–300, 1995.

[69] Jayadev Misra. A logic for concurrent programming: Safety. *Journal of Computer & Software Engineering*, 3(2):239–272, 1995.

[70] E. F. Moore. *Automata Studies*, chapter Gedanken experiments on sequential machines, pages 129–153. Princeton University Press, Princeton, New Jersey, 1956.

[71] L. J. Morell. Unit testing and analysis. Curriculum Module SEI-CM-9-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1988.

[72] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):7–30, July 1988.

[73] Joseph M. Morris. A theoretical basis for stepwise refinement and the progamming calculus. *Science of Computer Programming*, 9(3):287–306, 1987.

[74] Hanspeter Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer-Verlag, 1993.

[75] P. Naur. Revised report on algorithmic language Algol-60. *Computer Journal*, 5(349), 1960.

[76] George C. Necula and Peter Lee. Proof-carrying code. Technical Report CMU-CS-96-165, Canegie Mellon University, Pittsburgh, Pennsylvania 15213, September 1996.

[77] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 1995. Revision 2.0.

[78] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[79] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[80] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[81] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[82] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*, pages 46–57, New York, New York, 1977. IEEE.

[83] Arthur N. Prior. *Time and Modality.* John Locke lectures, 1955-6. Oxford, Clarendon Press, Oxford, 1957.

[84] Nicholas Rescher and Alasdair Urquhart. *Temporal Logic*, volume 3 of *Library of Exact Philosophy*. Springer-Verlag, New York, New York, 1971.

[85] G. Ricart and A. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, January 1981.

[86] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

[87] Beverly A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3:189–205, 1991.

[88] Sriram Sankar and Roger Hayes. ADL – an interface definition language for specifying and testing software. *ACM SIGPLAN Notices*, 29(8):13–21, August 1994.

[89] Sriram Sankar and Roger Hayes. Specifying and testing software components using ADL. Technical Report SMLI TR-94-23, Sun Microsystems Laboratories, Inc., M/S 29-01, 2550 Garcia Avenue, Mountain View, California 94043, April 1994.

[90] Roberto Saracco, J. R. W. Smith, and Rick Reed. *Telecommunications Systems Engineering Using SDL*. North-Holland, New York, New York, 1989.

[91] Sally Shlaer and Stephen J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Yourdon Press Computing Series. Yourdon Press, Englewood Cliffs, New Jersey 07632, 1988.

[92] Gowri Sandar Sivaprasad. Larch/CORBA: Specifying the behavior of CORBA-IDL interfaces. Master's thesis, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011-1040, November 1995. TR #95-27.

[93] Paolo A. G. Sivilotti and Peter A. Carlin. A tutorial for CC++. Technical Report CS-TR-94-02, Computer Science Department, California Institute of Technology, 256-80 Caltech, Pasadena, California 91125, 1994.

[94] Paolo A. G. Sivilotti and K. Mani Chandy. A distributed infrastructure for software component technology. Technical Report CS-TR-97-32, Computer Science Department, California Institute of Technology, 256-80 Caltech, Pasadena, California 91125, September 1997.

[95] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference.* MIT Press, 55 Hayward Street, Cambridge, Massachusetts 02142-1399, November 1995.

[96] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1991.

[97] Jan L. A. van de Snepscheut. On lattice theory and program semantics. Technical Report CS-TR-93-19, Computer Science Department, California Institute of Technology, 256-80 Caltech, Pasadena, California 91125, Spring 1993.

[98] Sreenivasa Rao Viswanadha and Sriram Sankar. Preliminary design of ADL/C++ – a specification language for C++. In *Second Conference on Object-Oriented Technologies and Systems (COOTS)*, June 1996.

[99] G. H. von Wright. And next. *Acta Philosophica Fennica*, 18:293–304, 1965.