# Compositional Verification of Calling Agent Hierarchies Using Templates

by

MARK ROGER HECKMAN

B.S. (University of California, San Diego) 1982
M.A. (University of California, San Diego) 1983
M.S. (University of California, Davis) 1992

DISSERTATION
Submitted in partial satisfaction of the requirements for the degree of
DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES
of the
UNIVERSITY OF CALIFORNIA
DAVIS

Approved:

[illegible]
Professor Karl Levitt (Chair)

[illegible]
Professor Ronald A. Olsson

[illegible]
Professor Venkatesh Akella

Committee in Charge

2000

**Compositional Verification**

**of Calling Agent Hierarchies**

**Using Templates**

## Abstract

Computers have been integrated into many different types of systems where the safety of lives and property directly depends on the correctness of the system. To have the maximum assurance that a system is correct, it should be mathematically proven that a system's implementation satisfies its specification, an often complicated and difficult process called *formal verification*. *Compositional verification* — independently verifying a system's components, and then "composing" the verified components into a complete system — may reduce the difficulty of verifying a large system, but may still be too difficult to be practical.

The goal of the work presented here is to simplify compositional verification by composing systems in a step-wise fashion, and by reusing specifications and proofs in each step. To this end, we have developed the *calling agent* model of interacting components and a standard specification style, based on a theory of composition developed by Abadi and Lamport, that ensures that specifications written for calling agents satisfy the requirements for composition.

We have also developed a method of viewing a system's calling agents as a hierarchy, which can be used to identify an optimal incremental composition strategy. An optimal strategy is one that composes only components whose composition results in an overall reduction in the complexity of the verification effort. We have developed an algorithm that finds an optimal composition strategy in calling agent hierarchies.

Our notion of a *template* encapsulates parameterized specifications and proofs. Instances of specific calling agents can be specified and composed simply by specifying the appropriate parameters. We have created an example template, developed using the HOL theorem proving system, and demonstrated how it can be applied to compose different calling agents simply by choosing different parameters.

Using our incremental composition method, once suitable templates have been created, an entire system consisting of many calling agents can be composed almost automatically. Examples of systems to which our method can be applied are micro-kernel operating systems, systems with nested security policies, and distributed systems such as the Domain Name System (DNS) and web caching systems.

To my parents, with love and appreciation.

# Contents

# List of Figures

## Acknowledgements

I would like to thank my advisor, Karl Levitt, for the unwavering support that he has given me during my years at U.C. Davis. Thanks also go to the other members of my committee, Ron Olsson and Venkatesh Akella, for their helpful suggestions as this dissertation grew close to completion.

Rob Lipton and Phil Nico were fonts of wisdom, experience, and perspective during the entire dissertation writing process. Their help, along with the love and encouragement of Kathy Loretz, other friends and family, and especially that of my parents, helped me more than I could ever express.

Portions of this research were funded by DARPA under contract USN N00014-93-1-1322 with the Office of Naval Research and by the National Security Agency's URP Program.

# Chapter 1

# Introduction

What is now proved was once only imagin'd.
*William Blake (1757-1827), "The Marriage of Heaven and Hell" ("Proverbs of Hell")*

Computers are used in an ever-increasing number of critical applications where errors in the computer system can cause a major loss of information, money, or even human life. These applications include communications networks, international financial systems, and fly-by-wire aircraft controls. The more critical the application, the more critical it is that the computer system be as flawless as possible.

An error in a computer system is an instance where the actual behavior of the system does not match the intended behavior of the system for some set of inputs. A *correct* computer system has no errors, i.e., its behavior always matches the intended behavior of the system for all inputs. The process of determining the correctness of a system is called *verification*.[1] Unfortunately, computer systems used in critical applications are almost always highly complex, and it may be extremely difficult — if not impossible — to show

[1] A separate issue from verification is determining whether or not a system satisfies the expectations of its users, a process called *validation*.

that they are correct.

There are two complementary approaches to determining the correctness of a system: *testing* and *formal verification.* Testing is an inductive approach. A limited number of test cases are supplied to the system and, if the system's behavior correctly matches its intended behavior for these test cases, the system is assumed (hoped) to be correct for all possible cases (to some probability). Formal verification is a deductive approach. The system and its specification are represented in a formal logic and, if the system can be mathematically proven to satisfy its specification, it is shown to be correct for all possible test cases [14].

Testing can never prove that a realistic, complex system is correct. The most rigorous set of tests can only show that whatever errors the system may yet contain remain undetected. Unlike testing, formal verification can, in principle, demonstrate that a system is correct, but formal verification nevertheless cannot replace testing. For one thing, formal verification can only show the correspondence between a system and its specification. A system may be proven to satisfy its specification, but that does not mean that the specification correctly described what the system is intended to do. Moreover, usability issues, in general, cannot be addressed by verification[35].

Furthermore, the greater the size and complexity of a system, the greater the size and complexity of the formal verification effort. A large and complex system may be, practically speaking, impossible to formally verify as a monolithic unit. Even when formal verification is possible, the potentially large cost to formally verify a system is a strong disincentive.

The purpose of the work described here is to increase the reliability — and safety — of computer systems by simplifying the formal verification process. We focus on the formal verification of large systems that consist of numerous components, each of which can be verified independently and then combined into the complete system. Our method, furthermore, is designed to scale well as the size and complexity of a system increases. The result of our work is a practical methodology that can reduce the verification effort by many orders of magnitude for some classes of systems.

## 1.1 Compositional Verification

All verification, whether testing or formal verification, boils down to comparing the implementation of a system against its specification. In testing, the specification is in terms of inputs and outputs: the outputs that are expected from the system as a result of a particular set of inputs. The implementation is run using each set of inputs to demonstrate that it generates the expected outputs.

Formal verification uses two specifications: a more detailed specification that represents the system implementation (the "low-level" specification) and a more abstract specification that represents the expected functioning of the system (the "high-level" specification). Because the low-level specification shows more details, it is called a *refinement* of the high-level specification. For example, when you present your claim check at a dry cleaner's, the clerk (hopefully) goes to the back of the shop, retrieves your cleaned clothes, and brings them out to you. All that you see is the clerk going to the back of the shop and returning with your clothes. The refinement of that high-level view includes all the details of how clothes are actually organized and stored in the back of the shop so that the clerk can quickly find them.[2]

### 1.1.1 Mechanical Theorem Provers

A formal proof can be an extremely tedious and time-consuming exercise. Even the simplest proofs can require thousands of mathematical steps and the large number of details make it easy to make mistakes. For this reason, programs have been developed that keep track of the proof details and that check the validity of each proof step. Such programs are called *proof checkers.* Another type of program, called *proof generators*, tries to automatically generate proofs. The general term for both proof checkers and proof generators is *mechanical theorem provers*, or just theorem provers, for short.

An example of a proof generator is the Boyer-Moore theorem prover[10], which uses a first-order logic. Most other mechanical theorem provers provide relatively little in the way of automatic proof generation tools, but offer a more powerful specification language and proof environment. An example of this type of theorem prover is HOL, a proof development system for higher order logic[16]. Higher-order logic is similar to predicate logic but also allows quantification over predicates and functions. Our verification methodology has been developed using HOL because of its expressive power and extensibility, but we could have used other theorem provers that also support higher order logic, such as PVS [31].

[2]Because the abstract specification is mathematically "cleaner" than the low-level specification, it might seem counter-intuitive to refer to the low-level representation as a "refinement." The term *reification* has been proposed as an alternative[21].

### 1.1.2 Refinement Step

The important step in formal verification consists of mathematically proving that the low-level specification satisfies the high-level specification. In a simple system, the proof may show that the relationship between inputs and outputs is the same for both specifications. In more complex systems, the proof may relate the states of the low-level specification to that of the high-level specification to demonstrate that the behavior of the implementation conforms to the system's expected behavior at all times. Because the proof shows that a low-level, refinement specification satisfies a high-level specification, it is often called a *refinement step*.

### 1.1.3 Compositional versus Non-compositional Verification

To cope with the size and complexity of formal verification for large systems, researchers have developed "divide and conquer" methods to separately specify and verify the components of a system, and then to prove that the components working together correctly implement the complete system. These methods are called *compositional* methods, because the system is composed out of its parts. A compositional proof of a system ideally can be less complex than a proof of the system as a monolithic unit because the internal details of each system component are hidden when carrying out the proof. The specification and proof of correctness for a component, furthermore, can be reused when the same component is used in a different system, reducing the overall cost of the compositional proof effort.

To see how compositional proof can hide details and reduce proof complexity, let us say that we are trying to verify the hardware that makes up the disk I/O subsystem of a computer system. The subsystem consists of a disk controller and a disk drive, which communicate via some well-defined interface. Each of the components consists of various circuitry, firmware, and other devices that, among other things, manage how the component deals with control and data signals on the interface. This is depicted in figure 1.1.

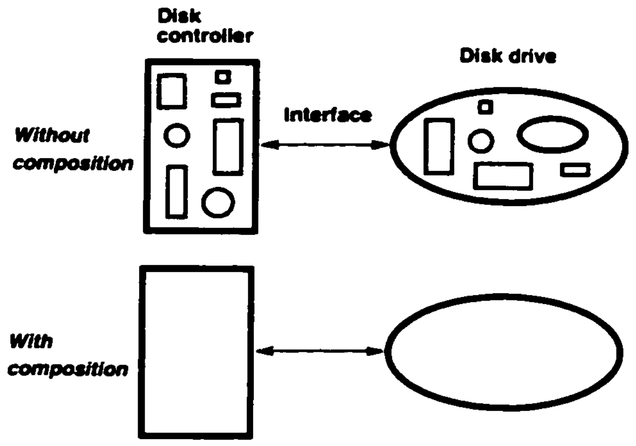

Figure 1.1: Compositional vs. non-compositional proof

If we compose this system as a monolithic unit, our proof must explicitly take into account all of the circuitry, firmware, and other devices that drive the interface within each component. When we do a composition proof, however, to compose this system we first verify that each of the components satisfies its own specification with respect to the interface. This requires us to prove that the various circuitry, firmware, and other devices within each component correctly put out, and respond to, control and data signals on the interface. Once we have done that, when we compose the controller and drive together, we can simply use the component specifications and no longer must concern ourselves with the devices of which each component is composed.

### 1.1.4 Reusing Specifications and Proofs

A device interacts with other devices solely through external interfaces. For this reason, once a device has been shown to satisfy its own abstract specification, the specification can be reused in the compositional proof of other systems in which that component is used. The proof that the device satisfies its specification does not have to be redone because the device's relationship to the external interfaces has not changed.

### 1.1.5 Composition Step

A compositional proof can show that two components, $A$ and $B$, working together, implement a larger system $AB$. At that point, the separate specifications for $A$ and $B$ can be replaced by the single specification for $AB$. While $A$ and $B$ may be verified independently, however, it must also be proven that they still work correctly in each other's presence.

To see why a verified component might work incorrectly when working in concert with another, consider the devices in the disk I/O system described above. Each component is verified with respect to its specification in terms of external inputs and outputs, but the specification is only valid under certain assumptions about the external inputs and outputs. For example, when data is being saved to the disk, the disk drive might assume that the controller will send it data in 512-byte blocks. If the controller does not satisfy this assumption and sends data in larger blocks, the disk drive will not correctly store the data.

To show that components will still work correctly when composed with each other, it must be proven that the externally visible behavior of each component satisfies the assumptions of the other components. This proof is called a *composition step*, to distinguish it from the refinement step.

The compositional verification methodology that we have developed is based on a refinement of the general compositional theory developed by Abadi and Lamport[3]. Their theory includes a composition step proof rule that describes the conditions under which a composition step proof is valid. A complete composition proof using Abadi and Lamport's method requires both a composition step and a refinement step.

Abadi and Lamport's composition theory can be tricky to apply. Their composition rule can be applied only provided that the specifications satisfy certain conditions, but because their specification method is so general it is easy to write specifications that do not satisfy the conditions. Our refinement of their method constrains the style in which specifications can be written. The regular, structured specification style in our method provides a framework in which reusable specifications and proofs can be written and ensures that specifications will satisfy the composition conditions. We will talk more about Abadi and Lamport's theory in chapter 3, and about our refinement of that theory in chapter 4.

## 1.2 Incremental Composition

In section 1.1.3, we explained how compositional proof hides the internal details of a component when the component is composed with other components. First, we verify that the component's implementation satisfies its specification for externally visible behavior (viz., the behavior visible at its interface); then, the implementation details can be "hidden" when the component is composed with other components.

But the internal devices that make up a component are themselves made up of even lower-level components. An integrated circuit (IC), for example, consists of various modules that are themselves made up of transistors. More and more frequently, ICs are formally verified with respect to the signals on their pins, which form the interface between a circuit and the circuit's environment. When we verify that the disk controller's implementation satisfies its specification for externally visible behavior, what we are really doing is composing the disk controller specification out of the specifications for its ICs and other components.

The composed disk I/O system, furthermore, is undoubtedly itself a component of a computer system. Once the disk I/O system specification is composed out of the disk controller and disk drive specifications, the disk I/O system specification can be composed with other components to form a complete computer system — or the computer system may itself be only a component of an even larger system. These different levels of composition are depicted in figure 1.2, which shows different levels of composition that might be used when composing an airplane.

At each level of composition, more and more detail is hidden within each component, which reduces the complexity of each compositional proof. Composition in different levels like this is called *incremental composition.* Each level of composition is called a *composition stage.*

### 1.2.1 Incremental Composition Strategies

In the airplane example of incremental composition described above, the different composition stages roughly corresponded to the granularity of the system components — circuits were composed with circuits, boards with boards, etc. Sometimes, however, it

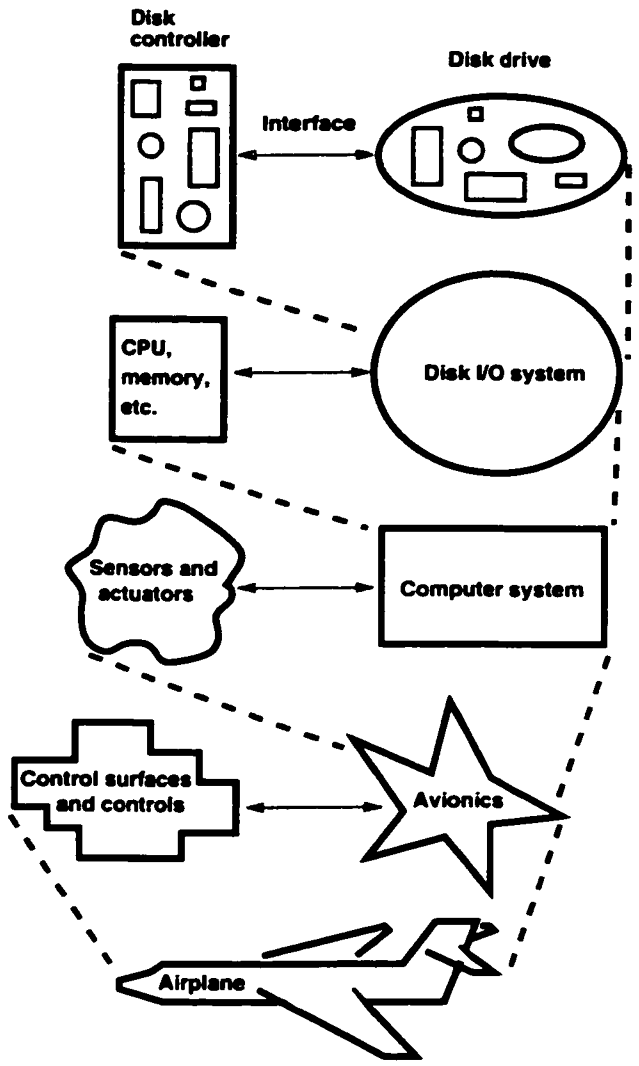

Figure 1.2: Composition stages

may be necessary to compose a large number of components of the same granularity. The larger the number of components to compose, of course, the greater the complexity of the composition proof. Given a large enough number of components, the composition proof may be too complex to be tractable.

To reduce the complexity of a composition proof, we need to look for strategies for applying incremental composition even when the composition stages are not obvious. In chapter 6, we introduce the concept of a *calling agent hierarchy* — a way of ordering components of the same granularity so that an incremental composition strategy can be found.

## 1.3 Proof Reuse

If a system contains many components, even incremental composition may not reduce the verification of the system to a manageable project. The problem is that potentially every composition proof stage is a highly complex proof in its own right. Although incremental proof can reduce the complexity of each stage, the total number of stages necessary to compose the entire system may be very large.

In some cases, however, one composition proof stage may resemble another so much that the proof for one stage can be reused, with only relatively slight modification, for the other. The effort to reuse a proof can be much less than the effort to create a proof "from scratch." In the best case, each of the composition proof stages will be quite similar, so that essentially the same proof can be applied over and over again. A practical, scalable compositional proof method seems to be possible only where significant proof reuse is possible, but the degree to which proofs can be reused depends to a large extent on the type of system that is to be composed. Specification and proof reuse is only possible where the components specifications are similar and the way that they interact with one another is fairly uniform.

### 1.3.1 Structural Homogeneity

When components in a system have similar specifications and interfaces, we say that they are ***structurally homogeneous.*** For example, the components of the airplane in the example described above are not particularly structurally homogeneous, but a multiprocessor system where every processor is the same as the others is extremely structurally homogeneous. Obviously, the degree of uniformity among components determines the degree to which a proof can be reused. Our work is directed at developing a reusable proof method for systems with a high degree of structural homogeneity.

Structurally homogeneous components, while they must have some degree of uniformity in their specification and interfaces, may, nevertheless, be quite different from one another. For example, the server processes in a micro-kernel operating system such as Mach [36] may have quite different functions, but they all use the same, basic communication system — message passing — and have similar interfaces. Although their functional specifications will differ, the specifications will, nevertheless have many structural similarities. The differences in their specifications can be abstracted and parameterized, so that the same abstract specification can be used to describe each of the server processes.

The degree of structural homogeneity, therefore, is determined by the extent to which the same abstract specification can be applied to describe each of the components.

Ideally, the specification for each component in a system is an instance of the same abstract specification.

### 1.3.2 Calling Agents

In chapter 5, we describe a model of structurally homogeneous components that interact with other components — the *calling agent* model. An example of a system of calling agents is the set of server processes in a micro-kernel operating system like Mach. The servers work in concert to implement system resource management policies; the composition of the specifications of the server processes together with the kernel should satisfy the specification for the operating system [19].

The calling agent model is also useful to describe components in a wide variety of other types of systems that are either distributed or where the implementation of system policies are distributed among a number of components. Examples include the Domain Name System (DNS) [29], web caching systems [6], and systems with nested security policies [18].

### 1.3.3 Templates

A proof can be reused in several ways. The least sophisticated method is to simply redo the proof, substituting labels and making modifications where necessary. A more sophisticated method is that described by Caplan and Harandi, where specifications and proofs are parameterized [11].

Parameterized specifications and proofs can be reused by instantiating the parameters with different values. The usefulness of this approach is that the parameters at different levels of abstraction can be values, types, programs, specifications, or even proofs themselves. Proofs and specifications that are parameterized in this way are said to be *generic*, because they can be instantiated in many different ways by a suitable choice of parameters.

For example, consider the composed disk I/O system discussed earlier. We could write a generic specification for a disk drive, parameterized by the disk capacity. In this way, we could reuse the generic specification to describe disk drives of any capacity. We could also parameterize other features, such as block size, to handle many different makes and types of drives. Similarly, a generic specification can be created to describe a variety of disk controllers. If we do the composition proof with respect to the generic specifications, the result is a generic proof that applies to any instances of the generic specifications.

We call the combination of generic calling agent specifications and generic composition proofs *templates*. Once a template has been created, it can be saved as a library in the context of a theorem-proving system. Proofs using templates can often be carried out automatically to a large degree by the theorem-proving system. Thus, incremental composition reduces the complexity of each compositional proof stage, and templates, because they can be reused, effectively reduce the number of compositional proof stages.

We have devised generic specifications for the calling agent model. A wide variety of components can be specified simply by choosing appropriate parameter values. We have also shown how to use generic (i.e., parameterized) composition proofs to compose the specifications. Our calling agent templates are covered in detail in chapter 6. An example template, with a detailed description of the generic specifications and composition proof, is given in chapter 7.

## 1.4 Contributions

The work described in this dissertation makes five important contributions:

1. Our refinement of Abadi and Lamport's composition theory results in a standardized and regular specification style that eliminates many possible sources of specification errors and simplifies the creation of compositional templates. The specification style constrains only the way that specifications are written, not the types of systems for which specifications can be written. Our refinement of Abadi and Lamport's theory is discussed in chapter 4.

2. Our calling agent model formalizes compositional reasoning about the components in distributed systems. The model is applicable to components in systems where resource management policies are distributed among a number of agents, such as micro-kernel operating systems and systems with nested security policies; and to distributed services, such as DNS and web caching. The calling agent model is discussed in chapter 5.

3. Our concept of templates turn the specification and compositional proof of distributed system components from a difficult and *ad hoc* activity into an engineering exercise that can be applied using well-defined rules. Templates are discussed in chapter 6.

4. Our calling agent hierarchy abstraction provides a way of organizing a large number of components into smaller groups so that an optimal incremental composition strategy

can be found. Our algorithm for finding an optimal incremental composition strategy in a calling agent hierarchy can be used to guide the application of templates to incremental composition stages. The optimal strategy simultaneously reduces the complexity of each composition stage but maximizes the benefit of each incremental stage in terms of hiding information, so that it holds the number of incremental stages to a minimum. Calling agent hierarchies and the incremental composition algorithm are also discussed in chapter 6.

5. We provide a fully worked example of a template, including generic specifications for calling agents and a generic composition proof. A detailed explanation of the example template is given in chapter 7.

Our work has been oriented toward developing a practical methodology for scalable, compositional verification based on proof reuse, applied to a generalized model of composable components. Our proof reuse strategy can significantly reduce the compositional proof effort for systems with structural homogeneity because proofs can be handled in a regular, largely automatic fashion by a theorem-prover.

## 1.5 Chapter Summaries

The remainder of this dissertation consists of the following chapters: Chapter 2 describes related work, including an overview of other work in proof reuse and compositional proof and refinement. We also include brief descriptions of other research topics that have directly contributed to our own. Chapter 3 describes the composition theory of Abadi and Lamport, while chapter 4 covers our refinement of Abadi and Lamport's method, including the specification style constraints and our mechanization of the composition theory in HOL.

Chapter 5 lays out our general model of a calling agent and gives a high-level description of how the composition method can be used to compose systems of calling agents. Chapter 6 explains the abstraction of a calling agent hierarchy and gives an algorithm for finding an incremental composition strategy. This chapter also further develops the concept of a template, showing how the basic calling agent model can be enhanced through an assortment of different parameters.

Chapter 7 is a detailed explanation of a HOL template that can be used to compose two calling agents and that demonstrates several of the calling agent model enhancements described in chapter 6. Chapter 8 summarizes this work and describes future directions it could take.

Appendix A contains the complete generic specifications for the example template and Appendix B contains two instantiations of the template.

# Chapter 2

# Related Work

Prove all things; hold fast that which is good.
*I Thessalonians 5:21, King James Version*

Our work draws from research in many different areas, including distributed systems, the application of formal methods to parallel programs, other specification and verification techniques including specification composition methodologies, other large-scale verification projects, and proof reuse. In this chapter, we will briefly discuss how these research areas have influenced our own research. We will also give an overview of selected other research projects in the areas of compositional proof and proof reuse, and compare and contrast our work with that of the other projects.

## 2.1 Systems of Calling Agents

The inspiration for the idea of calling agents as a general model of a composable system, and as an application of our composition method, were distributed systems that display a high degree of structural homogeneity. Systems of this type include the Domain

Name System (DNS), web caching systems, and distributed operating systems. Our model of calling agents is directly applicable to systems of this type. In this section, we describe the structural homogeneity in each type of system.

### 2.1.1 Distributed Operating Systems

An important class of distributed operating systems are those based on a micro-kernel design. A micro-kernel operating system has a small, fast kernel that implements the abstractions of processes and inter-process communication. Most operating system functions are implemented by special processes called *servers*. For example, files may be implemented by a file server process that calls a disk manager process. Examples of micro-kernel operating system designs include Mach [36], Synergy [33] and UC Davis's Silo [39].

In each of these operating systems, all server processes use the same communications interface and they share the same basic functionality of reading a message, taking some action, and sending a message in response. Their specifications, therefore, have a large degree of structural homogeneity when viewed abstractly.

The servers, furthermore, work together to implement system policies. In Mach, for example, files may be the responsibility of a file server, which creates the abstraction of files out of individual disk blocks. The file server depends on a disk server (or servers) to fetch and store disk blocks on the system's disk devices. In addition to these basic functions, each of the servers may implement buffering, to speed up access to frequently used files and disk blocks. The system policy implemented by the servers is to create the abstraction of files that are stored on disk devices, but this policy is a composition of the policies implemented by each of the individual servers.

The Synergy and Silo systems were designed to enforce a security policy that was implemented by one or more servers. The policy enforced by these systems must be a composition of the individual policies enforced by each of the servers, and is a natural application of our composition method [18].

### 2.1.2 Domain Name System

Another example of a system of calling agents is the Domain Name System (DNS). The DNS is a large, distributed database, used to translate between host names (e.g., "abc.rd.hp.com") and numeric Internet (IP) addresses (e.g., "192.151.52.10"). Programs called *name servers* each maintain the translation information for some local part of the Internet and make that information available to other parts. Programs called *resolvers* are used to query the name servers [29].

Host names are pathnames that contain a list of hierarchical domains and sub-domains. For example, the host name "abc.rd.hp.com" identifies a system "abc" in the "rd" sub-domain of the "hp" sub-domain of the "com" domain. This hierarchy can be depicted as a tree, as in figure 2.1.

Let us consider a DNS-like system where each of the name servers knows the IP addresses for the systems and name servers immediately "below" it. In our example, the *.com* name server knows the IP addresses for the *.hp* name server, as well as the *.ibm* name server and the *.aol* name server. The *.hp* name server knows the IP addresses for the *.rd* and *.sup* name servers below it, and the *.rd* name servers knows the IP address for the two systems *.abc* and *.xyz*.

If a resolver queries the *.com* name server for the IP address of the *.hp* name server,

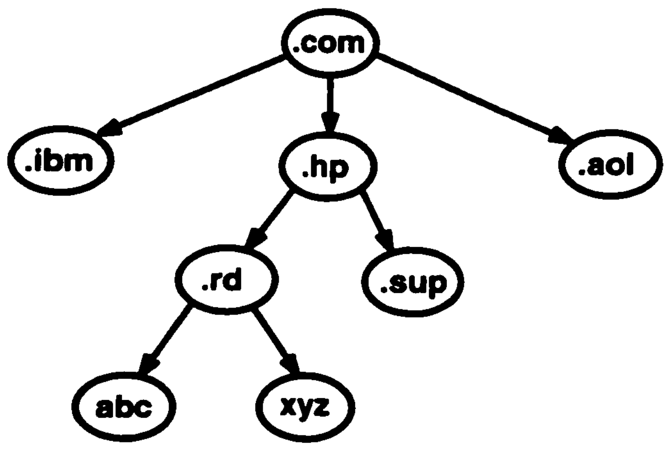

Figure 2.1: Model of hierarchical naming system

the *.com* name server simply returns that IP address. If, however, the query asks for the IP address of a system "beneath" the *.hp* name server, the *.com* name server can either refer the resolver to the *.hp* name server or else recursively query the *.hp* name server itself.

The recursive query model is a good example of a hierarchy of calling agents. The root of each sub-hierarchy calls the root of the next, until the name in the query is resolved to an IP address. The system is structurally homogeneous as well: although the name servers may be coded differently, their interfaces and functional specifications are essentially the same. Although many different name servers may be called upon to provide the IP address of a particular host, from the resolver's point of view it appears as if the entire database was available to the first name server that the resolver queried. The functional specification of the complete DNS-like system, therefore, is a composition of the functional specifications of all of its constituent name servers.

### 2.1.3 Web Caching Systems

Another example of a distributed service implemented by cooperating, structurally homogeneous agents is web caching. Web caching is a method of speeding access to web pages and reducing network congestion by storing copies of frequently accessed pages closer to the computers that are accessing the pages [6].

Like DNS, a network may contain many different individual web caching components. These components may be organized in a hierarchical fashion, or not. For example, as depicted in figure 2.2, there is a cache at the server site, an intermediate cache somewhere in the network between the server and the client, a local proxy cache for the client site, and a local cache for each client web browser.

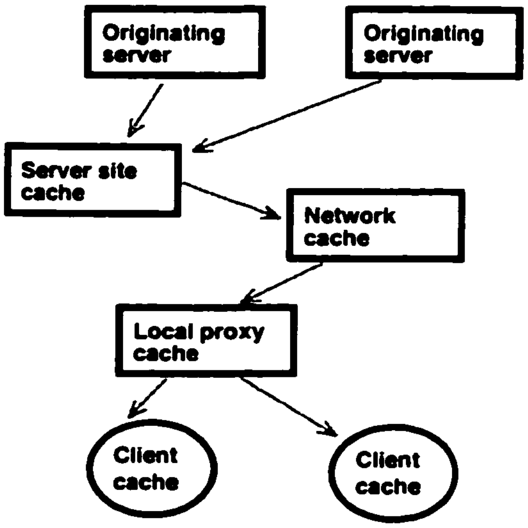

Figure 2.2: Web caching system

Unlike DNS, where each of the components implements the same functional specification of mapping domain names to IP addresses, each of the web caching components may implement a different web caching strategy. For example, the server site cache may emphasize the "freshness" of the copies that it provides, frequently updating the pages with the latest copies, while the local proxy cache may implement a "least recently used" policy of flushing pages from its cache. The overall web caching policy is a composition of the policies implemented by each cache component.

## 2.2 Formalisms for Parallel Systems

Distributed systems, like those described above, are typically concurrent: the components run simultaneously. Because the components run in parallel, it may not always be possible to determine in advance the order of events in the system. Systems with this property are said to be *non-deterministic.*

A common issue in concurrent, non-deterministic systems is called a *race condition*, where the results of an execution of the system vary depending on the order in which the component actions occur. For example, if we have two concurrent processes, one of which writes the value 1 to a particular memory location and the other that writes the value 2, the process that executes first will write first and the final value in the memory location will be the value written by the slower process.

It may be that the final value written to the memory location does not matter (in terms of the functional specification of the system) or it may be that we always want one process to write first and the other process second. In the latter case, the race condition could actually be an error condition. A specification and proof method used to verify distributed systems must be able to represent both parallelism and non-determinism in order to catch these kinds of error conditions.

### 2.2.1 Parallel Programs

The application of formal methods to parallel programs has been an active research area for many years. Hoare's CSP [20] and Misra and Chandy's UNITY [12] systems, for example, both provide proof methods to assist in the development and verification of parallel programs.

Both of these approaches, by design, consider parallelism and non-determinism, but they may be less suitable for the specification and verification of the type of distributed systems that we are interested in. For one thing, these methods are not designed to handle what are called *reactive* systems [17]. While most program verification is aimed at verifying systems that accept inputs, perform transformations on the inputs, and produce outputs, reactive systems continuously respond to external stimuli and maintain some kind of state, but may or may not produce outputs.

### 2.2.2 State-transition Models

A more general approach to modeling parallel systems is the state-transition method. In a state-transition representation of a system, the execution of a concurrent program or other parallel system is represented as a finite or infinite sequence of state transitions, as shown in figure 2.3.

$$\begin{tikzcd}
\boxed{s_0} \arrow[r, "e_1"] & \boxed{s_1} \arrow[r, "e_2"] & \boxed{s_2} \arrow[r, "e_3"] & \cdots
\end{tikzcd}$$

Figure 2.3: State transition sequence

The system changes from state $s_i$ to state $s_{i+1}$ as a result of some event $e_{i+1}$ (also called an *action*) caused by a component of the system. Events may be inputs, outputs, or changes to the internal state of a component, or combinations of the three.

Events are modeled as atomic actions, which means that they cannot happen simultaneously. This is a reasonable model of distributed systems, where an underlying mechanism enforces the independence of actions. For example, when modeling a message-passing system, we can model the sending and receiving of messages as atomic actions because the underlying system that implements the abstractions of messages ensures that messages are delivered as discrete objects. Even if two messages, $m_1$ and $m_2$, are simultaneously sent to the same receiver, they must arrive in the order $m_1, m_2$ or $m_2, m_1$. Messages sent to different receivers are similarly independent of one another, so that modeling their arrivals as atomic events is a reasonable abstraction.

The multiple possible interleavings of atomic events represents the concurrency of the system. For example, there are two possible orderings for the arrival of messages $m_1$ and $m_2$. If the order of arrival was deterministic, there would only be one possible ordering. Because concurrent systems are usually non-deterministic, however, it is often the case that many different sequences of events are possible.

The set of all possible event sequences (also called *execution traces* or *behaviors*) completely describes the functional behavior of a concurrent system. For this reason, the set of all possible traces can be used as a system specification.

It may seem that a specification that consisted of all possible execution traces of a system would be too detailed to be useful. This need not be the case. For example, we could specify as an invariant that a state variable $x$ was always equal to the value 5 using the following predicate:

$$\forall\ i.\ val\_x(get\_state(trace, i)) = 5$$

where *get_state* is a function that maps natural numbers to states in an execution trace, and *val_x* is a function that returns the value of the state variable $x$ in a state. This predicate is true for all traces in which the value of the state variable $x$ equals 5 in all states; we do not have to enumerate all the possible traces. The conjunction of many such predicates can be used to fully specify a system.

### 2.2.3 Safety and Progress Properties

Pnueli [32] and Lamport [25], among others, have used state-transition systems to specify systems in terms of both safety and progress properties. Intuitively, a safety property can be described as a specification that "nothing bad ever happens." A progress property can be described as a specification that "something good eventually happens" [3].

Because events and state changes in a trace are ordered in a sequence, both safety and progress properties are described in terms of temporal properties of the sequences. Safety properties can be defined in terms of conditions that hold in every state (or, alternatively, never hold). The invariant on the value of the state variable $x$, above, is an example of a safety property. Progress properties can be defined in terms of conditions that eventually occur, conditions that become true and remain true (henceforth), or in terms of the frequency with which the conditions occur. For example, we could specify a progress property on a state variable $y$ that says the value of $y$ eventually equals the value 6 using the following predicate:

$$\exists\ j.\ val\_y(get\_state(trace, j)) = 6$$

The conjunction of this predicate and the predicate on the value of $x$ that asserts that $x = 5$ in all states of a trace (from the previous section, 2.2.2) specifies all traces (and implicitly, therefore, all systems) where the value of $x$ is always 5 and the value of $y$ equals 6 at some point in the trace.

The state-transition model of a system, where the sets of traces describe all possible behaviors of the system, is an excellent way of modeling distributed, reactive systems. Not only does this model cleanly represent the concurrency and non-determinism of the system, but the use of safety and progress properties provides a convenient and succinct specification language.

## 2.3 Theorem Prover

The theorem prover that we have used in our specification and proofs is the Higher-Order Logic (HOL) theorem proving system, originally developed at Cambridge University [16]. The HOL system allows one to formally specify and prove theorems in a high order logic that allows quantification over predicates and functions. HOL has been widely used since 1988 for hardware verification. Entire microprocessors have been formally specified and verified using HOL, as well as smaller, special purpose digital logic devices [23, 13, 5]. HOL has also been applied to software verification [15, 40].

Drawbacks of HOL for use in an engineering environment include its rather slow speed and that it is primarily a proof checker, not a proof generator. An important advantage of HOL over other verification systems, however, is that proofs can only be developed in a controlled manner. HOL is based on five primitive axioms and eight primitive inference rules. All proofs generated using HOL can be reduced to one using only these basic axioms and inference rules, giving a high degree of assurance that the proofs are correct.

HOL is also a highly extensible system. In addition to its built-in theorem proving infrastructure, it is possible to extend the logic by creating new packages of definitions, theorems, and tools. A large user community has created many such packages.

HOL is designed to be used both as a stand-alone theorem proving tool and as an embedded theorem prover for customized verification systems. This makes it useful to us both in developing templates and to create an engineering tool in which templates can be applied.

## 2.4 Compositional Proof Methods

A variety of different compositional proof methods have been developed that use the state-transition model. Here we describe several of those methods.

Lam and Shankar developed a state-transition module composition method for layered modules [24]. Their method has three steps:

1. Decompose the system into a hierarchy of modules. Each module has an upper interface and a lower interface. The specification for a module is the upper interface.

2. Prove for all modules that, assuming that the environment of a module (see below) meets the input requirements of its upper and lower interfaces, the module correctly implements its upper interface specification.

3. Use a composition theorem to infer from the proofs in step 2 that the composition of a module and all modules beneath it implements the module's upper interface specification.

The environment for a module $M$ in Lam and Shankar's theory consists of the modules above $M$ that call $M$'s upper interface $U$, and the modules below $M$ to which $M$ calls through its lower interface $L$. Inputs to $M$ consist of inputs to $U$ (viz., values passed in calls from higher modules) and outputs from $L$ (viz., values returned from calls to lower modules). Similarly, outputs from $M$ consist of outputs from $U$ (viz., the values $M$ returns) as well as values passed by $M$ in calls to $L$. This model is depicted in figure 2.4.

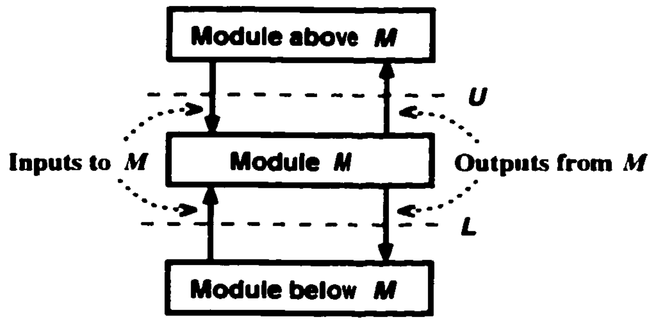

Figure 2.4: Lam and Shankar's module composition method

The composition method considers that $L$ is both the lower interface for $M$ and the upper interface for the module below $M$ (call it $N$), so outputs of $M$ to $L$ are simultaneously inputs to $N$, and inputs to $M$ from $L$ are simultaneously outputs from $N$. The composition theorem says that, if $M$ implements $U$ assuming that $L$ is implemented correctly, and if $N$ implements $L$, then the composition of $M$ and $N$ (and all the modules below $N$) implements $U$.

Jonsson developed a specification and composition method for distributed systems that communicate via asynchronous message passing [22]. In Jonsson's method, a component is specified using a standard model, called an *I/O-system*, where the state transitions are in terms of input and output events (i.e., sending and receiving messages). The specification for a component describes all possible orderings of input and output actions by that component. Purely internal events can also be represented by an event primitive. These specifications can be used to represent both safety and progress properties of the component.

Composition in Jonsson's method is essentially performed by matching up input and output events and removing them from the composed specification. For example, as depicted in figure 2.5, if we have two systems $A$ and $B$, and if $A$ has a single output event to $B$ while $B$ has a single input event from $A$, we can compose $A$ and $B$ into a single system where the sending of messages by $A$ to $B$ becomes an internal event.

Figure 2.5: Jonsson's composition method

Abadi and Lamport developed a general composition theory based on a formally defined principle for composing specifications [3]. Their composition principle recognizes that a system can be proven correct only under the assumption that the system's environment satisfies certain conditions. When composing a system, however, each of the components may be part of the environment of the others. The problem is that if a component $\pi_1$ is correct only assuming that component $\pi_2$ is correct, but $\pi_2$ is correct only assuming that $\pi_1$ is correct, it is not always necessarily true that the composition of $\pi_1$ and $\pi_2$ is correct. Abadi and Lamport's theory describes the conditions under which components can be composed. Because Abadi and Lamport's composition method is fundamental to our work, we will discuss it at greater length in chapter 3.

Jonsson's method, like that of Lam and Shankar, focuses on matching up input and output events. Unlike Lam and Shankar's method, however, Jonsson's method is not limited to hierarchical systems of modules, but can be used for composing distributed systems. Abadi and Lamport's composition method, unlike the other two methods, does not focus on matching inputs and outputs to produce a composed system, but treats input and output events like other state transitions. The focus is on showing that the state transitions of one component satisfy the environment assumptions of another. Their composition method is more flexible than the other two because the result of the composition is determined by the choice of abstract specification, whereas in the other methods the abstract specification is explicitly derived from the modules and composition method. Thus, Abadi and Lamport's method, like Lam and Shankar's method, can be applied to the composition of hierarchical systems of modules, and like Jonsson's method, Abadi and Lamport's method can be applied to the composition of distributed, communicating modules.

We have used Abadi and Lamport's method as a basis for our work. Like Jonsson, however, we use a standard model of a component, but within Abadi and Lamport's framework. The standard component model and standard specifications provide a regular, structured specification style that simplifies the tasks of developing and composing specifications, and makes the proof effort more amenable to automation.

## 2.5 Specification and Proof Reuse

Possibly the most complete example of a verified, composed system to date is the "CLI short stack," built by researchers at Computational Logic, Inc. [8]. The system consists of a verified compiler, assembler, linker and microprocessor. (An operating system kernel, called "Kit", was also verified, but was not part of the stack because it had been designed and verified to run on a different processor [7].) The system was a stack in the sense that each of the layers was designed and verified as an abstract machine built on the next lower layer [30]. Although CLI used a similar specification model for each layer of the stack, their specifications and proofs were not parameterized and so each proof was essentially unique.

Windley verified a microprocessor, AVM-1, using a model of an abstract machine that was similar to CLI's, but developed it into a theory of abstract theories in HOL [38]. The abstract theories consist of generic specifications and theorems about them that can be instantiated for each level of abstraction in a hierarchical proof. Abstract theories can also be used to define generic proofs by laying out the complete set of lemmas that must be proven in order to complete a refinement step between two levels of abstraction.

The verification methods used by Windley and CLI, though they were effective for the systems that they verified, are not composable. A system cannot be specified as components and then the component specifications be easily combined, and it is difficult to see how functions could be added to the Kit operating system or AVM-1 microprocessor, for example, without redoing their entire proofs. Windley's and CLI's systems were not distributed, so they did not have to deal with concurrency. Schubert later addressed the concurrency problem by creating an interpreter calculus that can be used to compose interpreter specifications [34].

Although the model of abstract theories developed by Windley and Schubert is parameterized, their method is not fully parameterized in the sense that the proofs themselves can be reused and thereby automated. Their abstract theories are essentially libraries of theorems about the specifications. Our idea of templates, although similar to Windley's and Schubert's abstract theories, are designed to fully encapsulate the reusable specifications and proofs so that they can be applied as "turn-key" proof modules.

Aagaard and Leeser demonstrated a method for reusing proofs of hardware components [1]. Their method is based on parameterized modules. In addition to parameterizing by structural variables such as signal bit-widths, however, their method provides a measure of composability by parameterizing submodules. Given a module that consists of several submodules, the module can be shown to be correct for all possible structural variables as well as all correct implementations of the submodules.

Like Aaraard and Leeser's approach, our method uses parameterized modules and structural variables. Unlike their method, however, we use a general model of a component, and the parameters include not only structural variables but also variables that determine the function of the module.

Caplan and Harandi gave a high-level description of a logical framework for verification of reusable software components [11]. Their framework supports the reuse of parameterized specifications and proofs. The parameters at different levels of abstraction can be values, types, programs, specifications, or even proofs themselves.

While they sketched out their framework, Caplan and Harandi did not demonstrate it on more than small examples and their work did not address compositional proof. Our work applies Caplan and Harandi's general ideas of proof reuse to a compositional framework, and extends the idea of reusability by encapsulating the specifications and proofs into templates within the context of a theorem-proving system, making some degree of automation possible.

## 2.6 Summary

Our work draws on research in a number of different areas. Our notion of templates was anticipated by Windley's work with abstract theories in HOL. Unlike Windley's abstract theories, however, our templates are intended to be fully encapsulated specifications and proofs, rather than a library of theorems about generic specifications. Our method of proof reuse has been most influenced by Caplan and Harandi's framework for reusable components. Unlike in their framework, however, we have applied their basic concepts of reusable specifications and proofs to compositional proof, and with an eye toward applying the parameterized proofs in the context of an automated engineering application.

The template-based method for composing complex systems is predicated on a refinement of the composition method of Abadi and Lamport. Because of the importance of Abadi and Lamport's work to our own, we discuss their method and our refinement of it in the following chapters.

# Chapter 3

# The Composition Method of Abadi and Lamport

We prove what we want to prove, and the real difficulty is to know what we want to prove.
*Emile Chartier (1868-1951), in "Webster's Electronic Quotebase", ed. Keith Mohler, 1994*

Our composition method is based to a large extent on a refinement of the compositional proof method developed by Martín Abadi and Leslie Lamport. Abadi and Lamport provided a composition principle and proof rule for composing modular specifications that consist of both safety and progress properties [3]. Their composition method is based on the *transition-axiom* specification method [26] and a refinement mapping method of proving that one specification implements another [2]. In this chapter, we summarize the main concepts and terms used in their transition-axiom, refinement mapping and composition methods. We will refer to these concepts and terms in subsequent chapters. Except where indicated, the material in this chapter has been summarized from reference [3].

## 3.1 Composition Principle

The goal of composition is to prove that a composed system satisfies its specification provided that all of its components satisfy their specifications. Each component, however, interacts with other components and with the outside world, which make up the component's *environment*. The specification of what a component does may only be guaranteed provided that the component's environment satisfies certain assumptions.

For example, a component $A$ in a message passing system may be guaranteed to respond to messages in the order that they are received provided that the rest of the system can only append new messages to the tail of $A$'s mailbox. If the environment could insert messages at other locations in the mailbox, delete messages from the mailbox, or rearrange messages in the mailbox, then $A$ could not be guaranteed to satisfy that specification.

Let us say that there is a second component, $B$, that also depends on the environment leaving its mailbox alone (except for appending new messages to the end of the mailbox). $B$ is part of $A$'s environment, so $A$ depends on $B$ to not interfere with $A$'s mailbox. But $A$ is part of $B$'s environment, so $B$ depends on $A$ to not interfere with $B$'s mailbox. In general, in order to be able to compose a set of components into a complete system, each of the system's components and the system's environment must satisfy each of the other components' environment assumptions. Abadi and Lamport informally define this *composition principle* as follows:

*Let $\Pi$ be the composition of $\Pi_1, \ldots, \Pi_n$, and let the following conditions hold:*

1. *Every Component $\Pi_i$ guarantees $M_i$ under environment assumption $E_i$.*
2. *The environment assumption $E_i$ of each component $\Pi_i$ is satisfied if the environment of $\Pi$ satisfies $E$ and every $\Pi_j$ satisfies $M_j$.*
3. *$\Pi$ guarantees $M$ if each component $\Pi_i$ guarantees $M_i$.*

*Then $\Pi$ guarantees $M$ under environment assumption $E$.*

The composition principle is depicted in figure 3.1 for two components, where $E$ is the composed system's environment assumption and $M$ is the composed system specification, $E_1$ and $E_2$ are the environment assumptions for the two components, and $M_1$ and $M_2$ are the component specifications. As shown in the figure, the conjunction of $E$ and $M_2$ must satisfy the environment assumption $E_1$, and the conjunction of $E$ and $M_1$ must satisfy the environment assumption $E_2$.

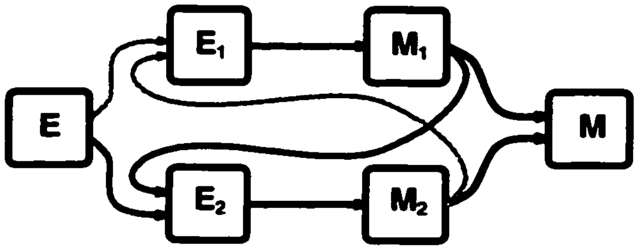

Figure 3.1: Composition principle

As shown in the figure, the composition principle's reasoning is circular because each $M_i$ holds only when $E_i$ holds, but we assume every $M_i$ holds when proving that every $E_i$ holds. Abadi and Lamport's chief result is a proof rule that clearly sets out the conditions under which the reasoning is valid, despite the circularity.

## 3.2 Behaviors

In Lamport's transition-axiom method [26], systems are specified as sequences of atomic state transitions, called *behaviors*, $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots$, where each $s_k$ is a state ($s_0$ is the initial state) and each $\alpha_k$ is an *agent*. State transitions occur due to actions taken by agents. Behaviors may be finite or infinite, but a finite computation can be represented by an infinite sequence if the sequence reaches a point where the state no longer changes. A state is a binding of values to a set of state variables that represent only the visible interface of a system, not internal variables.

A sequential program would have only one agent (the program) and only one possible correct behavior for a given set of inputs (i.e., the series of state transitions due to the execution of program steps). A general specification for the sequential program would consist of the set of all allowed behaviors for all possible inputs (e.g., depending on the inputs, different conditional branches might be taken), where the inputs are specified as part of the initial state of a behavior.

A concurrent system, however, consists of more than one agent, has potentially many possible interleavings of events and, therefore, has many possible state transition sequences. For example, a system with two concurrently running agents, $\alpha_1$ and $\alpha_2$, each with a single possible atomic action, $a_1$ and $a_2$, has two different possible interleavings of events: $a_1, a_2$ and $a_2, a_1$. The nondeterminism of the system is represented by the different possible event orderings.

A ***$\mu$-stuttering step*** is a transition from a state $s$ to the same state $s$ (i.e., the state does not change) where the agent is a member of the agent set $\mu$. Let $\natural_\mu \sigma$ be the behavior obtained from a sequence $\sigma$ after replacing every sequence $s \xrightarrow{\alpha_1} s \xrightarrow{\alpha_2} s \ldots$ $(\alpha_i \in \mu)$ of $\mu$-stuttering steps with the single state $s$. Two behaviors $\sigma$ and $\tau$ are considered to be *stuttering equivalent* if $\natural_\mu \sigma = \natural_\mu \tau$. If we consider non-stuttering transitions to be changes to the observable system state, then stuttering steps would correspond to invisible changes to the internal state of the system.

Stuttering steps are necessary when proving that an implementation satisfies a high-level specification. The implementation, as a refinement of the specification, may contain many details that do not appear in the specification. Thus, some state changes in the implementation may be "filtered" by the refinement function, so a state transition in the implementation may map to a stuttering step in the specification.

## 3.3 Properties

A set of behaviors that is closed under stuttering (i.e., that contains all behaviors that are stuttering equivalent to behaviors in the set) is called a *property*. As discussed previously in section 2.2.3, there are two types of properties: safety properties and progress properties. Intuitively, safety properties of a system define the acceptable initial states and the allowable state transitions. Progress properties assert that specific state transitions eventually occur.

The specification of a system consists of the conjunction of various safety and progress properties. If, for example, $I$ is a state predicate that determines the set of valid initial states (interpreted as the property consisting of all behaviors whose initial state satisfies the predicate), $T(N)$ is a property that consists of all behaviors whose every state transition is either a stuttering step or else is allowed by a next-state relation $N$, and $L$ is a progress property, then the specification for a program could be defined as the property $I \cap T(N) \cap L$, which means the set of behaviors whose initial state satisfies $I$, whose every state transition is a stuttering step or satisfies the next-state relation $N$, and that satisfies the progress requirement on the scheduling of transitions.

For example, consider the concurrent system described above, with two agents $\alpha_1$ and $\alpha_2$, each with a single possible action, $a_1$ and $a_2$. As noted earlier, it has two different possible interleavings of events: $a_1, a_2$ and $a_2, a_1$. To specify this, we could write a property where $I$ is always true (i.e., there are no constraints on the initial state), where $N$ allows only the transitions $a_1$ and $a_2$ and allows each to happen only once, and where the progress property asserts that $a_1$ and $a_2$ will both eventually happen. Because of stuttering steps, an infinite number of possible behaviors satisfy the property, but they all have the general form $[x^*, a_1, x^*, a_2, x^*]$ or $[x^*, a_2, x^*, a_1, x^*]$, where $x^*$ means zero or more stuttering steps by either or both agents. Without the progress property, our specification would also allow the additional behaviors $\emptyset$ (i.e., nothing happens), $[x^*]$ (nothing but stuttering steps happen), $[x^*, a_1, x^*]$ and $[x^*, a_2, x^*]$ (only one of the transitions happen).

### 3.3.1 Machine Closure

It is important that a progress property constrain the scheduling of state transitions but does not constrain which state transitions can occur. Otherwise, it may be impossible to build a system that satisfies the specification. Abadi and Lamport provide the following example of why this is so [3].

The initial state of a program and the computer instructions that the program executes are determined by the program's code, which specifies the program's next-state relation. Consider the next-state relation, depicted in figure 3.2, for a program with a single state variable $x$. The initial state predicate asserts that $x$ initially equals 0. The state transitions from the initial state that are permitted by the next-state relation are to

$x = 1$ and $x = 2$.

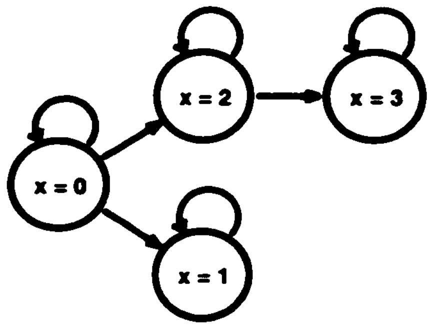

Figure 3.2: Next-state relation

Now assume that a progress property in the specification for the program asserts that $x = 3$ at some time during the execution of the program. For the program to satisfy that property, it must never make the transition to $x = 1$. In effect, this progress property implies a further safety property — that $x$ never equals 1. A compiler could not simply use the program's code (i.e., next-state relation) to generate an executable file that satisfied the specification. but would somehow also have to deduce that one of the allowed transitions must not occur.

Abadi and Lamport define a pair $(M, P)$ of properties to be *machine-closed* if $P$ does not imply any safety properties not implied by $M$ [2]. For specifications written in the form $I \cap T(N) \cap L$, where $I$ and $T(N)$ are safety properties and $L$ is a progress property, we want the pair $(I \cap T(N), I \cap T(N) \cap L)$ to be machine-closed so that we can only specify a system that is possible to implement.

The closure of a property $Q$, denoted $\overline{Q}$, is the smallest safety property containing $Q$. The closure of a property $P \cap L$, where $P$ is a safety property, $L$ is a progress property, and $(P, P \cap L)$ is machine-closed, would simply be $P$.

## 3.4 System versus Environment

The inputs to a system come from outside the system, which is the system's *environment*. For example, when we specify a server process in an operating system, the server's environment consists of the kernel and all other processes (both other servers and user processes). The universe of agents is divided into two sets of agents, the system agents and the environment agents.

A specification for a system could be written that accounts for the behavior of the system for all possible inputs. It is simpler, however, to specify a system assuming that the inputs satisfy some definition of correctness. A specification written this way must take into account that the system is only expected to work correctly if the environment works correctly (i.e., the environment provides correct inputs). The server processes in an operating system, for example, cooperate to implement the overall operating system specification. It would be simpler to specify one server assuming that the other servers also conform to their specifications rather than specifying the behavior of the server in all possible conditions. The specification of a system $M$ and the assumptions about its environment $E$ is the property $E \Rightarrow M$, which includes all behaviors where the system satisfies its specification or the environment does not.

### 3.4.1 Machine Realizability

A specification of the form $E \Rightarrow M$ asserts only that the system specified by $M$ behaves correctly when the environment satisfies the assumptions $E$, not that the environment will always satisfy $E$. If $M$ also asserted that the environment always behaved in a certain way, then the system would be impossible to implement correctly because no matter how you built the system you could not control the environment. Such a specification is said to be *unrealizable.*

A realizable specification is a specification for which it is possible to build a *correct implementation.* A correct implementation allows only behaviors that are permitted by the specification. A specification may have both realizable and unrealizable parts. Because a correct implementation may only allow behaviors that are also permitted by the specification, it can only implement the realizable part of the specification.

A *receptive* property is a property that is completely realizable (i.e., all behaviors in the property have a correct implementation). Abadi and Lamport define a pair of properties $(M, P)$ to be *machine-realizable* if the pair is machine-closed and if $P$ is receptive. Because a system consists of a set of agents $\mu$, a machine-realizable system specification is said to be *$\mu$-machine-realizable.*

### 3.4.2 Writing the Environment Property as a Safety Property

An interesting result of Abadi and Lamport is that when an environment specification is machine-realizable, the environment progress property can be incorporated into the system's progress property, and the realizable part of the resulting specification is equivalent to the original.

For an environment specification $I \cap E_S \cap E_L$, where $I$ is an initial state predicate, $E_S$ is the environment safety property, and $E_L$ is the environment progress property, and a system specification $M_S \cap M_L$, theorem 1 of reference [3] says that if $(E_S, E_S \cap E_L)$ is $\neg\mu$-machine-realizable (where $\mu$ is the set of system agents and $\neg\mu$ is the set of environment agents), then

$$I \cap E_S \cap E_L \Rightarrow M_S \cap M_L$$

and

$$I \cap E_S \Rightarrow M_S \cap (E_L \Rightarrow M_L),$$

are $\mu$-equirealizable (i.e., they have the same realizable parts).

One of the chief requirements for valid application of the Composition Principle is that all environment assumptions are safety properties. This theorem ensures that even if the environment specification contains a progress property, we can put the complete system specification into a form suitable for applying the Composition Principle.

## 3.5 Internal versus External State

State variables are used to completely describe the interface between the system and the environment. The system, however, may also have internal state that is not modifiable by the environment. Abadi and Lamport "hide" internal state using existential quantification. The property that specifies a system with internal state must also assert that the internal state has correct initial values and that only the agents in the system (i.e., not the environment agents) can modify the internal state. The complete safety property $M_s$ for a system with agent set $\mu$ and with internal state $x$ can be written in this form:

$$\exists x : I_x \cap TA_{\neg\mu}(U_x) \cap TA_{\mu}(N)$$

where $I_x$ is a predicate on the initial values of $x$; $TA_{\neg\mu}(U_x)$ is an assertion that, for all state transitions caused by the environment agents, the internal state remains unchanged; and

$TA_\mu(N)$ is the next state relation for the system. The complete specification for a system, including the system's progress property, can be written in the form $\exists x : M_s \cap M_L$.

## 3.6 Proof Rule

We give here the formal statement of Abadi and Lamport's composition proof rule from reference [3]:

*If $\mu_1$, $\mu_2$, and $\mu_1 \cup \mu_2$ are agent sets and $E$, $E_1$, $E_2$, $M_1$, and $M_2$ are properties such that:*

1. *$E = I \cap P$, $E_1 = I_1 \cap P_1$, and $E_2 = I_2 \cap P_2$, where*
   (a) *$I$, $I_1$, and $I_2$ are state predicates.*
   (b) *$P$, $P_1$, and $P_2$ are safety properties that constrain at most $\neg(\mu_1 \cup \mu_2)$, $\neg\mu_1$, and $\neg\mu_2$, respectively.*
2. *$\overline{M_1}$ and $\overline{M_2}$ constrain at most $\mu_1$ and $\mu_2$, respectively.*
3. *$\mu_1 \cap \mu_2 = \emptyset$*

*then the rule of inference*

$$\frac{E \cap \overline{M_1} \cap \overline{M_2} \subseteq E_1 \cap E_2}{(E_1 \Rightarrow M_1) \cap (E_2 \Rightarrow M_2) \subseteq (E \Rightarrow M_1 \cap M_2)}$$

*is sound.*

This rule easily generalizes for systems of $n$ components ($n > 2$). Strictly speaking, the conclusion of Abadi and Lamport's rule of inference applies only to the realizable parts of the specifications, but the specifications that we use in our approach to describe calling agents are machine-realizable so the conclusion shown here is equivalent.

## 3.7 Refinement

Our remaining step to complete the composition is to find a refinement mapping from $M_1 \wedge M_2$ to $M$.

### 3.7.1 Refinement Mappings

A refinement mapping is a function from states in one specification (e.g., $M_1 \cap M_2$) to states in another specification (e.g., $M$). Intuitively, we can think of $M_1 \cap M_2$ as the implementation of a system and $M$ as the abstract specification for the system.

Abadi and Lamport describe four conditions that must be satisfied by a refinement mapping $f$:

1. $f$ must preserve the external state (i.e., the mapping is a total function from the external state and implementation internal state to the abstract specification internal state).

2. $f$ must map initial states in the implementation to initial states in the abstract specification.

3. $f$ must map state transitions in the implementation to state transitions in the abstract specification.

4. $f$ maps behaviors that satisfy the implementation (including environment steps) to behaviors that satisfy the abstract specification's progress properties.

The refinement step in a composition proof is to find a suitable refinement mapping between an implementation and a specification and to prove that it satisfies the four conditions.

After applying the composition rule to a set of component specifications, the result is an implementation specification that we can use in the refinement step. After proving that our refinement mapping from the implementation to the abstract system specification is correct, the composition proof is complete.

# Chapter 4

# Mechanizing the Composition Method in HOL

That which you hear you'll swear you see, there is such unity in the proofs.
*Winter's Tale, Act V. Scene 2*

Abadi and Lamport's composition method is extremely general. Their semantic model of composition permits the proof developer to use his or her own specification format and is intended to be independent of any particular specification language or logic. This kind of flexibility is powerful, but it can make the development of specifications somewhat *ad hoc*.

One risk of *ad hoc* specification development is that a developer might create specifications that do not satisfy the conditions for the composition rule. For example, a developer might easily write a specification that is not machine-realizable by specifying progress properties that also imply safety properties.

When specifications are hand-crafted as needed, furthermore, they will likely be so different from one another that proof reuse is difficult or impossible. Reusable proofs require some kind of standardized specification format.

The importance of composing only specifications that satisfy the composition rule conditions, and the need for standardized specifications, has led us to develop a refinement of Abadi and Lamport's method. Our refinement puts some limits on the way that specifications can be written, but does not limit the generality of the types of systems that can be specified and reasoned about using Abadi and Lamport's method.

While our refinement satisfies our requirements for "safe" and standardized specifications, many other different refinements that have the same effect could be developed. Ours was designed to make it easier to specify and reason about the types of distributed systems that are of interest to us.

In this chapter, we describe our refinement of Abadi and Lamport's semantic model and our implementation of it in HOL. We note here that we do not derive Abadi and Lamport's results, but use the composition principle as an axiom.

## 4.1 A Specification Language for Composition

Abadi and Lamport's work was intended to be independent of any particular specification language or logic. They point out, however, that a logic for composition must have the following attributes [3]:

1. The logic must express sets of behaviors as properties (i.e., the sets must be closed under stuttering).

2. The logic must have an attribute that Abadi and Lamport call *explicitness*. This

means that determining if a behavior satisfies some formula can only be based on comparing the state variables that are free variables in the formula (the behavior may have other free variables that are irrelevant for this purpose). Conversely, it means that if some formula says that one state variable is modified, but no other state variables are even mentioned in the formula, then every state variable must implicitly be a free variable in the formula.

The second attribute can be tricky to implement. Abadi and Lamport say that a practical language must let you implicitly include some variables as free variables, even though it does not appear explicitly in the formula. The suggestion here is that a practical language allows you to implicitly say that "any state variables not shown here remain unchanged."[1]

## 4.2 Traces

We call behaviors "traces" (as in "execution traces"). Traces are defined as functions from positive integers to $\langle agent, state\rangle$ pairs, called "trace elements."

For some trace *trace* and a number $i$, $trace(i)$ is the trace element that corresponds to agent $\alpha_i$ and state $s_i$ in the behavior represented by *trace*. The state in $trace(0)$ (the first trace element) is the initial system state. (The agent in $trace(0)$ is undefined because there is no preceding state from which to transition to the initial state.)

Abadi and Lamport define properties as sets of behaviors that are closed under

[1]Although Abadi and Lamport intended that their work be independent of any particular specification language or logic, we note in passing that Lamport's temporal logic of actions language [26] satisfies both of these requirements.

stuttering. This could be represented in HOL (using the HOL "sets" library [28]) as

$$\{(trace : num \rightarrow trace\_element) \mid P(trace)\},$$

the set of functions that map natural numbers to trace elements that satisfy a predicate $P$, where $P$ allows stuttering steps. Practically speaking, however, we never need to use the set notation in applying the composition principle, but use only the predicate.

Following Abadi and Lamport, we distinguish safety properties from progress properties and safety properties on the initial state from safety properties on state transitions. We use the "standard" form

$$I \cap TA_{\mu}(N) \cap P$$

for specifications, to ensure realizability.

## 4.3 Internal State

Abadi and Lamport "hide" internal state using existential quantification. The meaning of the existential operator in Abadi and Lamport's approach is that there exists an entire sequence of values for the internal state — call it the "internal trace" — that corresponds to a behavior (and therefore allows stuttering). Their approach also requires an assertion that the environment does not modify the internal state. A benefit of their approach is that all references to the internal state, including the assertion that the environment does not touch it, are encapsulated in the system property. No reference to the internal system state appears in the externally visible (interface) state or in the environment property. A HOL implementation of some property $Q$ of a system that had internal state might look like the following:

$$\forall\ trace.\ \exists\ itrace.\ Q(trace, itrace),$$

where *trace* is a function from numbers to interface state, *itrace* is a function from numbers to internal state, and $Q$ includes the assertion that any transitions in the trace by non-system agents leave the internal state of the system unchanged.

In our formulation, we tried another approach to implementing internal state. This approach extends the definition of system state in a trace element to include, but maintain separately, internal as well as external state. In other words, where Abadi and Lamport's existential quantifier is used to define a sequence of internal state transitions that is parallel to the sequence of external state transitions in the trace, our approach merges the internal state into each trace element. Traces are therefore functions from natural numbers to the triple $\langle agent, (global\ state, internal\ state)\rangle$. The type of the pair $(global\ state, internal\ state)$ is called a *system state*, because it completely describes the entire state of the system.

Because a trace element contains both the global and internal state, our method does not hide the internal state; Abadi and Lamport's assertions that the environment does not modify the internal state become part of the safety properties of the environment instead of the system. When we compose systems and prove that the environment assumptions of each are satisfied, we must at the same time prove that each system does not modify the others' internal state. This is, in general, a very simple proof.

One effect of our approach is that it helps our logic satisfy the explicitness criterion. A trace element contains the value for every state variable, both global and internal, so when traces or trace elements are free in a formula, so are all of the state variables.

## 4.4 Properties

A property is a predicate that defines a set of traces. For example, consider the property

```
(∀ trace . example_property trace =
  (∀ i . val_x(trace i) = 5))
```

where *val_x* is a function that extracts the value of the state variable $x$. This property is true for all traces where the state variable $x$ has the value 5 at all times.

By universally quantifying traces in properties, the traces and their constituent trace elements are free variables in the properties. Because trace elements are $\langle agent, state\rangle$ pairs and because states are $\langle global\ state, internal\ state\rangle$ pairs, all state variables are free in properties. In this way, our logic satisfies the explicitness requirement.

We use the Abadi and Lamport's "standard" form for the properties that define a system:

$$I \cap TA_\mu(N) \cap P$$

where $I$ is the initial state property, $TA_\mu(N)$ is the system's next-state relation, and $P$ is the progress property. In this section, we describe our implementation of these three types of properties.

### 4.4.1 Initial State Properties

The general form of an initial state property is

```
(∀ trace . initial_property trace = I(trace 0))
```

where `I` is a predicate on the initial state. Note that this definition implicitly allows stuttering steps because it cares only about the initial state, and so is a true property.

There is an initial state safety property for the external state, the "environment initial state property", and another initial state safety property that applies to the internal system state, the "system initial state property".

## 4.4.2 State Transition Properties

A state transition property specifies the next-state relation for the system. It therefore is the disjunction of all allowable state transitions for a particular agent set, including stuttering steps.

State transition properties have this general form:

```
(∀ trace. transition_property trace =
 (∀ i. let ss1 = get_trace_state(trace i) and ss2 = get_trace_state(trace(i+1))
  in ((get_trace_agent ss2) = α) ⟹
  (ss2 = ss1) ∨ (ss2 = transition_function_1 ss1) ∨ ... ∨(transition_relation_n ss1 ss2)))
```

where `get_trace_state` and `get_trace_agent` are functions that extract the system state and agent, respectively, from each trace element. The predicate compares the states in adjacent trace elements; the difference between the two states represents a state transition. Because a next-state relation applies to state transitions by a particular agent set, the predicate only compares states in transitions caused by a particular agent $\alpha$ (which here represents an agent set).

This predicate is true for all traces where every state transition caused by agent $\alpha$ is either a stuttering step or else satisfies one of the valid atomic state transitions (the predicate is trivially true when the agent is not $\alpha$). By explicitly permitting stuttering steps, we make our state transition specifications closed under stuttering and, hence, properties. The atomic state transitions may be expressed as relations or functions from states to states.

#### 4.4.2.1 Specifying State Transitions

There are two approaches to specifying how a system is allowed to transition from state to state. One approach emphasizes what a system is permitted to do; anything not specifically permitted is forbidden. The other approach emphasizes what a system may not do; anything not explicitly forbidden is permitted. We use both approaches in our specifications.

Atomic state transitions are predicates on pairs of states, for example $P\ s_1\ s_2$. The predicate is satisfied if the second state represents a valid transition from the first state.

**State Transition Functions** A special case of an atomic state transition is the predicate $s_2 = F(s_1)$, which is true only when the state $s_2$ is equal to the value calculated by the function $F$ applied to state $s_1$. $F$ is an example of a *state transition function*, which has this general form:

```
(∀ ss : system_state . state_transition_function ss =
  ¬(precondition ss) ⇒ ss | (next_state_n ss))
```

A state transition function specifies what a system is allowed to do. If some precondition is not met, the only permitted transition is a stuttering step. If the precondition is met, the next allowable state is calculated by the function.

For example, consider a system where the state consists of two state variables, $a$ and $b$. The following would be a state transition function written in HOL that modified state variable $a$ provided that state variable $b$ met some precondition, and that leaves $b$ unchanged.

```
(∀ ss : system_state . state_transition_function ss =
 let a = (get_a ss) and b = (get_b ss) in
% If b does not meet the precondition then return state unchanged. %
  ((~precond b) → ss |
% Otherwise, modify a... %
  (let new_a = (modify a) in
% construct the new state %
  (put_a new_a ss))))
```

(Note that a HOL comment is delimited by a pair of % characters.) This state transition function uses destructor functions to extract state values from the *system_state* data type. For example, *get_a ss* extracts the value of the state variable $a$. The constructor *put_a* replaces the value of $a$ in the state, leaving all other state variables unchanged. The syntax *precondition* $\rightarrow$ $cond_1|cond_2$ is a conditional statement that returns either *cond_1* or *cond_2*, depending on whether the *precondition* is true or false, respectively.

An advantage of expressing state transition functions in this way is that state variables not explicitly changed by the function are implicitly left unchanged, which simplifies the transition specification because we do not have to exhaustively assert whether or not each variable changes. The function above, for example, has no constructor functions that modify the value of the state variable $b$, so its value is unchanged.

**State Transition Relations** State transition functions emphasize what the transition is permitted to do. In cases where a large number of possible state transitions must be defined, a specification may be more succinct if it instead emphasizes what a system may not do. A *state transition relation* can be used for this purpose.

Here is an example state transition relation for the system described above that has two state variables labeled *a* and *b*. The relation permits (i.e., the value of the predicate is true for) any transition that leaves the state variable *a* unchanged, but there are no constraints on how *b*'s value may change:

```
(∀ ss1 ss2 : system_state . state_transition_relation ss =
 let a1 = (get_a ss1) and a2 = (get_a ss2)in
% check that a has not been changed %
  (a1 = a2))
```

## 4.5 Progress Properties

We specify progress properties using the following general form:

```
(∀ trace . progress_property_n trace =
  (∀ i . ∃ j . i≤j ∧
   (get_system_state(trace(j+1)) =
     transition_n(get_system_state(trace j)))))
```

where `transition_n` is a state transition function. This property is true for all traces where at any point `i` in the trace there is another point `j` that occurs after `i` and where the transition will happen. In temporal logic (e.g., see [27]), this statement is represented as "always eventually `transition_n`", or $\Box\Diamond transition\_n$. There is one progress property for each state transition. The complete progress property for a system is the conjunction of the individual state transition progress properties.

Progress properties of this type imply that all traces are infinite, but that does not imply that only infinitely running systems can be specified. As previously discussed in section 3.2, a finite computation can be represented by an infinite sequence if the sequence reaches a point where the state no longer changes. That point is reached in systems specified using this type of progress property when none of the state transition preconditions are satisfied and they are not satisfied at any point thereafter in the trace.

The progress property for a system is the conjunction of the individual state transition progress properties. The system progress property asserts that each transition will occur infinitely often. Note that a state transition defaults to a stuttering step when the transition's preconditions are not met, so it is possible that the state change defined by the transition will never occur unless the precondition becomes true at some point and remains true until after the transition has occurred. Our progress properties, therefore, implement a "weak fairness" scheduling policy on transitions [4].

We chose this form for progress properties for several reasons. First, a progress property in this form asserts no new safety properties. Abadi and Lamport use the term *machine-closed* to described specifications where the progress properties have this characteristic. A machine-closed specification should make it simple, for some system $M$, to create $\overline{M}$, the smallest safety property that contains $M$, as required by the antecedent of the composition proof rule. We merely take out the progress properties, leaving the initial state and state transition properties.

Another reason for using this general form for progress properties is that it simplifies the mapping of some kinds of transitions. For example, it simplifies the mapping of any transitions of an implementation that never map to stuttering steps in the abstract specification. In such a case, it is easy to prove that, because the implementation transition always eventually happens, the abstract specification transition also always eventually happens. Transitions that always map to stuttering steps, of course, are even easier to map.

## 4.6 Composition

Having specified the properties of each of the component systems, the next step in composing the system is to prove the antecedent of Abadi and Lamport's proof rule (summarized in section 3.6).

The antecedent of the rule for composing $n$ components becomes the following HOL proof goal:

```
(∀ trace .
  ((env_init trace) ∧ (env_transition trace) ∧
   (sys_1_init trace) ∧ (sys_1_transition trace) ∧ ... ∧
   (sys_n_init trace) ∧ (sys_n_transition trace)) ⟹
  ((env_1_init trace) ∧ (env_1_transition trace) ∧ ... ∧
   (env_n_init trace) ∧ (env_n_transition trace)))
```

where `env_init` and `env_transition` are the overall environment initial state and state transition properties (whose conjunction corresponds to $E$ in the proof rule), `sys_1_init` and `sys_1_transition` are the initial internal state and state transition safety properties for component system 1 (whose conjunction corresponds to $M_1$ in the proof rule), and `env_1_init` and `env_1_transition` are the environment initial state and state transition properties for component system 1 (whose conjunction corresponds to $E_1$ in the proof rule).

### 4.6.1 Proof Complexity

The composition proof can rapidly grow in complexity as a function of the number of state transitions on the left-hand side of the implication and the number of systems that are being composed. For each system state transition property *sys_x_transition* on the left side of the implication and environment state transition property *env_y_transition* on the right side of the implication ($x \neq y$, because the agent sets of a system and its environment do not intersect), we must prove that every transition in *sys_x_transition* implies at least one of the transitions in *env_y_transition*.

In general, if we are composing $n$ modules and, on average, each of the modules has $m$ state transitions, then we must prove that each of the $nm$ state transitions satisfy $n-1$ environments. In the worst case, we have to do $\Theta(n^2m)$ proofs. In practice, however, we can often use incremental composition to reduce the number of modules — and the resulting number of transition proofs — in each composition proof stage.

## 4.7 Refinement

We define a refinement mapping $f$ as a function that corresponds to Abadi and Lamport's $f^*$ formula [3]. The function takes as input a system state of the form $(t, y)$, where $t$ is the external state and $y$ represents all of the internal state of the components, and returns as output a system state in the form $(s, x)$, where $s$ is the external state of the composed system and $x$ is the internal state of the composed system. Because in our refinement of Abadi and Lamport's method we have merged the external and internal state, our refinement mappings are simply HOL functions of type *component_state* $\rightarrow$ *composed_system_state*.

### 4.7.1 Mapping the Initial State

The goal for proving that the initial state in the components maps to the initial state in the composed system is

```
(∀ ss . I ss ⟹ I' (f ss))
```

where f is the refinement mapping function from ss, a system state of the components, to a system state of the composed system; I is the conjunction of the environment and individual system initial state predicates of the components; and I' is the conjunction of the environment and system initial state predicates for the composed system.

### 4.7.2 Mapping the State Transitions

The next step in the refinement is to prove the mapping of each of the state transitions of the components to either a state transition or a stuttering step in the composed system. Because the specification of state transitions may implicitly assume that the system state is always in some kind of consistent state, it may not be the case that the component transitions map up to the composed system transitions for all possible values in the range of the state variables. Invariants are used to eliminate unreachable states from the proof of the mapping. The proof that the state transitions map up may, therefore, require the proof of the invariants as an obligation.

A goal to prove an invariant P has the form

```
(∀ trace i .
  ((env_init trace) ∧ (env_transition trace) ∧
  (sys_init trace) ∧ (sys_transition trace)) ⇒ P(trace i)
```

which means that P holds for all states in a trace assuming the initial states and state transitions defined for the environment and component systems. The invariant becomes a theorem that is used in mapping the state transitions.

The HOL goal for mapping state transitions is

```
(∀ trace i .
  ((env_init trace) ∧ (env_transition trace) ∧
  (sys_init trace) ∧ (sys_transition trace)) ⇒
    (N' (f'(trace i)) (f'(trace(i+1))) ∨
    ((f'(trace i)) = (f'(trace(i+1))))))
```

where `N'` is the next state relation for the composed system and `f'` maps trace elements. (The agent in a trace element, however, is unchanged by the mapping.) This means that the safety properties of the components must map up for every state transition to either a state transition or stuttering step in the composed system. The antecedents of this goal imply the invariants, which can then be used in the proof.

### 4.7.3 Mapping the Progress Properties

The goal to prove that the progress properties of the components imply the progress properties of the composed system has this form:

```
(∀ trace .
  (I trace ∧ TE trace ∧ TM trace ∧ L trace) ⇒ L' (f'' trace))
```

where `I` is the initial state property for the components, `TE` is the state transition property for the environment, `TM` is the state transition property for the components (which consists of the conjunction of the individual component system state transition properties), `L` is the progress property for the components (i.e., the conjunction of the individual progress properties for all the components), `L'` is the progress property for the composed system, and `f''` applies the refinement mapping function to every element in a trace.

## 4.8 Summary

In this chapter, we have described our refinement of the composition method of Abadi and Lamport and our mechanization of their semantic model in HOL. Our refinement structures the development of specifications but does not otherwise limit the generality of Abadi and Lamport's method.

The goal of providing this structured specification method is to ensure that specifications that are to be composed satisfy the requirements of the composition rule and to simplify the use of the specifications in compositional proofs. The "standard" specification style can be applied in the development of specifications for our standard model of a component in a distributed system, the *calling agent* model, which is described in the following chapter. In chapter 7, we give a detailed example of a composition proof that uses our method.

# Chapter 5

# Calling Agents

We should recoil, stricken with sorrow and shame,
To see disclosed, by such dread proof, how ill
That which is done accords with what is known
To reason.

*William Wordsworth, "The Excursion", V:254*

Our composition method is applicable to systems that consist of a collection of structurally homogeneous components — components that have similar specifications and interfaces — where the specifications and proofs used to compose a small group of components can be reused over and over again until the entire system has been incrementally composed. The ability to reuse specifications and proofs ensures that our method scales well as the size and complexity of a system increases: no matter how many additional components are added to the system, the work to develop the specifications and proofs has only to be done once.

In order to reuse specifications and proofs, some standard model of a component must be used. We have devised a standard model of a component in a distributed system that satisfies the need for structural homogeneity but that is sufficiently extensible to be useful for specifying a wide variety of systems and their components. We call this standard model a *calling agent.*

An example of a calling agent system that was described in section 2.1.1 is the set of server processes in a micro-kernel operating system. In a micro-kernel operating system, the server processes that make up the system are relatively homogeneous. This homogeneity suggests that the same, basic specification can be used to describe each of the server processes, and that similar proofs can be done to show their correctness. The differences between the functions performed by each server process can be parameterized, so that the specification for each server process is an instantiation of the same, general, server process specification.

In this chapter, we describe the calling agent model, explore different variations on the basic calling agent model, and introduce how the composition principle can be used to compose a system of calling agents.

## 5.1 Calling Agent Model

We use a general model of autonomous agents that communicate through asynchronous message-passing. Agents call other agents by sending *request* messages. An agent that receives a request performs some action on behalf of the agent that sent the request and then returns a *response* message. The actions performed by an agent that responds to a request may include calculating a function value, retrieving or modifying system state, or sending request messages to other calling agents. Request messages may contain call parameters and response messages may contain return values. Figure 5.1 depicts this calling agent model.

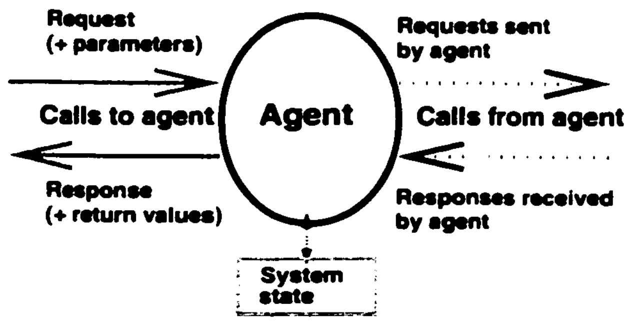

Figure 5.1: Abstract model of a calling agent

The model is based on the assumption that an underlying system implements the abstractions of agents and message-passing. For example, the underlying system could be a micro-kernel that implements processes and mailboxes, or a network that carries messages between nodes.

An agent that is called may in turn call other agents before returning a response, so that a call to one agent may result in a cascade of calls between agents. The cascade of all possible calls from one agent to another can be represented as a directed graph. The nodes and links in the directed graph are static in the sense that all components — and the components that they can call — must be known in advance. As we will discuss later (see section 6.2.2), in some cases calling agents may call only a subset of the agents that they can call, depending on different parameter or system state values.

## 5.2 Varieties of Calling Agents

Calling agents can be classified according to the following criteria:

1. *Whether or not they call other agents.* An agent that does not call other agents, which we call a *terminal agent*, becomes a leaf in the call graph. Non-terminal agents may call more than one other agent.

2. *Whether or not they maintain part of the functional system state.* In our basic calling agent model, each calling agent has its own mailbox that is part of the global system state and has local state to manage the messages that the calling agent receives. A calling agent may also maintain additional local state as part of the agent's function. For example, a file server might maintain a directory of files and a buffer cache of frequently accessed disk blocks. We differentiate between the state that is used to manage messages, which all calling agents have, and the functional state, and refer to an agent that maintains no functional state as a *stateless calling agent.*

### 5.2.1 Stateless Calling Agents

The functional system state may be distributed among many calling agents, but there is no requirement that a calling agent must maintain part of the functional system state. A stateless agent may implement a mathematical function on its request message parameters, in turn calling other agents or returning a response value without calling other agents.

For example, consider a secure operating system that has a layered access control policy [18]. There may be a hierarchy of security policies, $P_1, \ldots, P_n$, where $P_i$ has priority over $P_j$ for all $i < j$. Each policy may be implemented in a different server process. This system is depicted in figure 5.2.

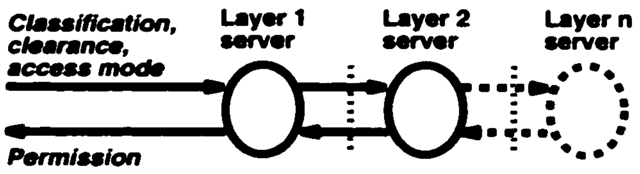

Figure 5.2: Security layers as server processes

A security policy server accepts as parameters a user's clearance, a classification for some system resource or object, and an access mode, and applies a boolean function to determine if the user is permitted to access the resource or object in the requested way. If a security policy server denies access, it returns a value of *FALSE* in its response message. If the server grants access, it sends the clearance, classification, and access mode to the next-lower priority server, and so on down to the lowest priority security policy server, which is a terminal server that returns *TRUE* or *FALSE*.

A server that has approved access returns the response value from the next-lower server as its own response. A user can only be granted access permission if all of the security servers grant access permission; a user will be denied access if any of the servers deny permission.

### 5.2.1.1 Varieties of Stateless Agents

The simplest calling agent manages no functional system state nor calls other agents. It simply accepts parameters in input requests, calculates a mathematical function of the input parameters, and returns the function value in the response message. This simple type of agent is depicted in figure 5.3, where the agent, passed the parameter $x$ in the request message, returns the value $f(x)$ in the response.

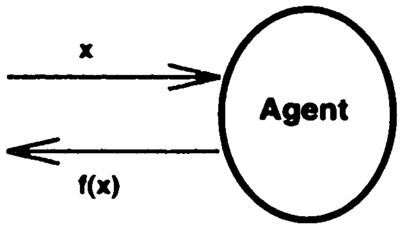

Figure 5.3: Simple calling agent

A calling agent that calls other agents may calculate a number of different functions as part of the processing that it does. For example, consider an agent $A$ that accepts a single parameter in a request message and that itself makes a single call to another agent $B$. As shown in figure 5.4, agent $A$ may calculate a function $a_1$ on the input value before sending a request message to $B$, and may calculate a function $a_2$ on the value returned by $B$ before sending a response message. Both $a_1$ and $a_2$ may be the identity function.

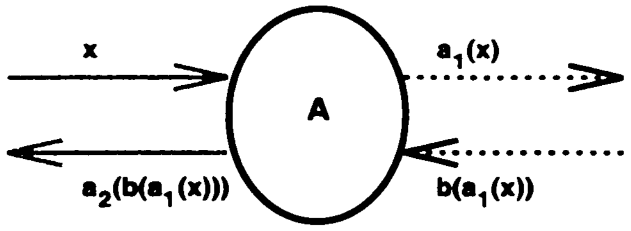

Figure 5.4: Calling agent pre- and post-functions

### 5.2.2 Agents that Maintain State

The functional state of a system may be distributed among different system components, which in our model of a system are calling agents. An agent that reads and returns values from its part of the functional system state is very similar to an agent that simply calculates a mathematical function value. Instead of calculating a function value, however, this variety of agent maps an input value $x$, which designates a portion of the agent's local state $s$, onto the corresponding state value $s(x)$. The agent may also update the functional state that it maintains, using the value passed in a request message. A calling agent that maintains functional state is depicted in figure 5.5.

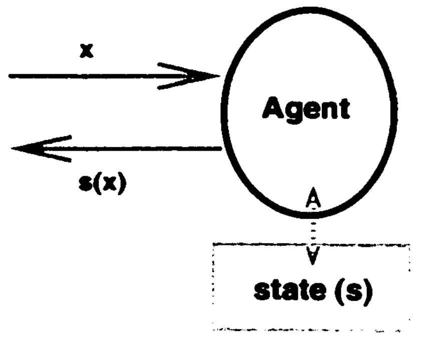

Figure 5.5: Agent retrieving a state value

An agent that maintains functional state may call other agents that maintain state. For example, a file server would maintain a directory of files and a buffer cache of frequently accessed disk blocks and would call a disk manager to allocate and deallocate disk blocks.

## 5.3 Reasoning About Calling Agent Systems

The goal of reasoning about a system of calling agents is to prove that a correct response message is generated for any request message sent to the system. We can prove this by composing a system of calling agents into a single calling agent, as shown in figure 5.6, where the system consisting of calling agents $A$ and $B$ is composed into the system $AB$.

In our model, every agent has its own mailbox, to which other agents send request messages. A called agent responds to a request by sending a message to the requesting

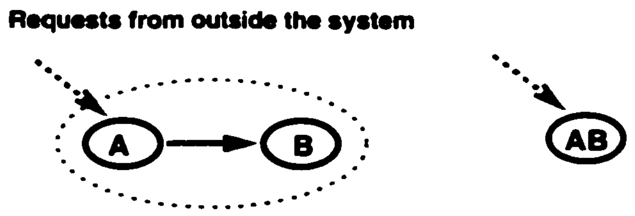

Figure 5.6: Composing a simple calling agent system

agent's mailbox. The global state of the system consists of all the mailboxes.

A calling agent in our model corresponds to an agent in Abadi and Lamport's specification method. As depicted in figure 5.6, from the point of view of an agent $B$ the sending of a request message by another agent $A$ is an action of $B$'s environment. Similarly, from the point of view of agent $A$, the sending of a response message by $B$ is an action of $A$'s environment. After composing $A$ and $B$, the composed system will consist of agents $A$ and $B$, and the environment will consist of all other agents.

### 5.3.1 Calling Agent Atomic Transitions

A call from one calling agent $A$ to another calling agent $B$, where $B$ is a terminal calling agent that does not call other agents, consists of three atomic state transitions:

1. $A$ sends a request message to $B$'s mailbox
2. Eventually, when $A$'s message reaches the front of $B$'s mailbox (i.e., after $B$ has read all of the messages in the mailbox ahead of $A$'s message), $B$ reads $A$'s message and moves it to $B$'s internal queue
3. $B$ processes the request and sends a response message to $A$'s mailbox

The three state transitions are depicted by the arrows in figure 5.7.

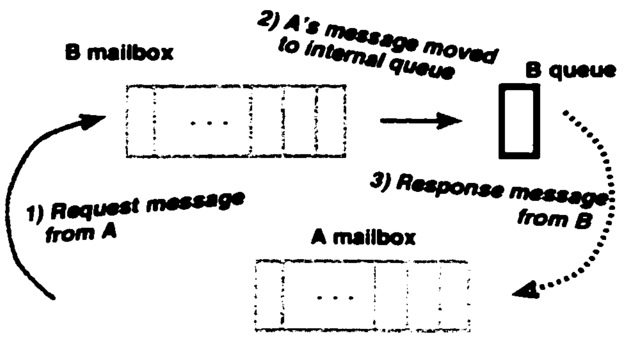
B mailbox
2) A's message moved to internal queue
B queue
1) Request message from A
3) Response message from B
A mailbox

Figure 5.7: Calling agent state transitions

### 5.3.2 Environment Atomic Transitions

In addition to a stuttering step, the environment of a calling agent has two transitions:

1. the environment appends a message to the calling agent's mailbox (the *env_send* transition), or
2. the state changes in some other way that does not affect the calling agent (the *env_arbitrary* transition). The *env_arbitrary* transition is written as a relation, not as a function, because a calling agent is specified to work correctly as long as the environment behaves correctly with respect to the agent's mailbox only. The relation simply says that the agent's mailbox remains unchanged by the transition, but there is no restriction on how the rest of the global state (other mailboxes) can change.

## 5.4 Specifying and Composing Calling Agents

In this section, we give simple descriptive specifications for some of the varieties of calling agents that we mentioned above and give a high-level overview of how the composition method can be applied to compose some simple call graph structures. In chapter 7, we will give a fully worked example of composing calling agents.

### 5.4.1 Terminal Calling Agents

A terminal agent does not call any other agents. The specification of a terminal calling agent has three transitions: ***agent_reads***, ***agent_responds***, and ***stutter***. Progress properties, described in section 4.5, ensure that these transitions occur infinitely often.

Here is a descriptive specification for the ***agent_reads*** transition function:

```
∀ system_state.  agent_reads system_state =
  If no messages in the mailbox then
    return system_state
  else
    return the system state modified as follows:
      transfer message at head of mailbox to tail of internal queue,
      leaving all other mailboxes and head of internal queue unchanged
```

If there are no messages in the mailbox when an ***agent_reads*** transition occurs, the state transition function implements a stuttering step by returning the system state unchanged. If there is at least one message in the mailbox, the transition function modifies the system state by moving the message from the head of the mailbox to the tail of the internal queue.

A descriptive specification for the ***agent_responds*** transition resembles that of the *agent_reads* transition in that the system state is modified only if a precondition is satisfied:

```
∀ system_state.  agent_responds system_state =
  If no messages in internal queue then
    return system_state
  else begin
    return the system state modified as follows:
      read one message from the head of the internal queue;
      let $src$ be the ID of the message sender;
      let $data$ be the request data value;
      put a response message in $src$'s mailbox that contains $f(data)$
  end
```

If there are no messages in the internal queue when the transition occurs, the state transition function implements a stuttering step. If there is at least one request message in the internal queue, the *agent_responds* transition function applies the function $f$ to the data value in the request message, and returns that calculated value in a response message.

#### 5.4.1.1 Composing Terminal Calling Agents

Let us say that we are composing the two terminal agents $A$ and $B$ shown in figure 5.8 into the composed system $AB$. Because $B$ never sends a request message to $A$, $A$ never sends a response message to $B$. Thus, when some other agent sends a request message to $A$, which appears as an *env_send* transition to $A$, the transition appears to $B$ as an *env_arbitrary* transition by the environment (because $B$'s mailbox is unchanged).

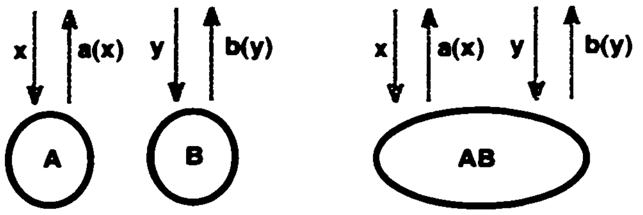

Figure 5.8: Composition of $A$ and $B$ to $AB$

Similarly, any *agent_responds* transition by $A$ sends a message to a mailbox other than $B$'s and appears to $B$ as an *env_arbitrary* environment transition. An *agent_reads* transition by $A$, which moves a message from $A$'s mailbox to its internal buffer, also appears as an *env_arbitrary* environment transition to $B$. The view from $A$ with regard to $B$ is symmetric.

Thus, when applying the composition rule (see section 3.6), it is straightforward to show that the state transitions in the overall environment $E$ and the safety properties of $A$, $\overline{M_A}$, satisfy the state transitions in $B$'s environment specification $E_B$ (and that the state transitions in $E$ and $\overline{M_B}$ satisfy the state transitions in $E_A$).

#### 5.4.1.2 Refinement Step

In general, the refinement step of a composition of two components $A$ and $B$ into the system $AB$ shows that

$$f(A \land B) \Rightarrow AB,$$

where $f$ is a refinement function that maps states in the components to states in the composed system. When we compose two terminal agents, however, the composition does not hide any state — neither the mailboxes that make up the global state nor the internal state for each agent — and it does not hide any of the state transitions. Request messages to $A$ still go to $A$'s mailbox, and request messages to $B$ still go to $B$'s mailbox. Thus, the refinement function for the composition of two terminal agents is essentially the identity function.

### 5.4.2 Simple, Non-terminal Calling Agents

Non-terminal agents call other agents. The simplest type of non-terminal calling agent makes a single call to another agent.

Let us say for example that a non-terminal agent $A$ receives a request message from an agent in $A$'s environment, called $E$. Furthermore, let us say that in order to process the request from $E$, $A$ must call a terminal agent $B$. After receiving a request message from $E$,

$A$ sends a request message to $B$'s mailbox and cannot send a response to $E$ until it gets a response from $B$. Once $B$ has responded to $A$, $A$ can send a response message to $E$. This system is depicted in figure 5.9. As shown in this figure, $E$ sends some value $x$ in a request message to $A$, $A$ sends the value of the function $a(x)$ to $B$, $B$ returns to $A$ the function value $b(a(x))$, and $A$ returns to $E$ the value that $A$ received from $B$.

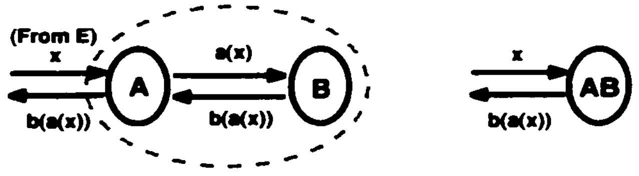

Figure 5.9: Composition of non-terminal and terminal agents

For now, we assume that $B$ receives request messages from $A$ only. We will talk about reusing agents in a hierarchy (i.e., in the call graph there are multiple edges to the same node) in chapter 6.

The goal of composing $A$ with $B$ is to show that the composed system satisfies a specification for a single terminal calling agent, call it $AB$. The composition proof must show that $AB$ will accept the same request messages and return the same response messages as $A$ working together with $B$.

In our model, each agent has only a single mailbox, and it must process messages in the order that they are received. This means that $A$ must continue to process messages while it is waiting for $B$ to respond, so $A$ must buffer the original request message from $E$ until it receives $B$'s response.

Meanwhile, $B$ reads the request message from $A$, processes it, and sends a response message back to $A$'s mailbox. Eventually, due to the progress properties that ensure that

$A$ processes messages in its buffer and mailbox, the message sent by $B$ reaches the head of $A$'s mailbox, so $A$ reads the message, pulls $E$'s original request message out of the buffer, and sends a response message to $E$.

The messages that $A$ continues to read while it is waiting for a response from $B$ may be either additional requests from $E$ or responses from $B$ to previous requests from $A$. If a message is a request from $E$, $A$ generates a new request message for $B$ and buffers $E$'s request message. If a message is a response from $B$ to an earlier request from $A$, $A$ matches the response from $B$ with the original, buffered request from $E$ and sends a response.

The different transitions necessary for $A$ to respond to a request are shown pictorially in figure 5.10. The solid arrows represent the original request message to $A$ and $A$'s request message to $B$. The dotted lines depict the path of $B$'s response to $A$ and $A$'s ultimate response.

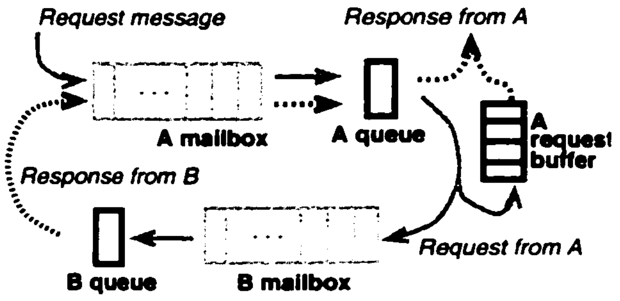

Figure 5.10: Message path in simple calling agent chain

Because mailboxes are FIFO queues, a terminal agent is guaranteed to respond to requests in the order that the requests arrive. This means that $A$ will receive responses from $B$ in the order that it sent requests to $B$ and that $A$'s own responses will occur in the same order that it received requests.

#### 5.4.2.1 Specifying Simple, Non-terminal Calling Agents

The *agent_reads* transition for the simple, non-terminal calling agent is the same as for a terminal agent. The *agent_responds* transition, however, is quite different. The agent must determine the origin of the message and take different actions depending on whether the message was a request from another agent or a response from the terminal agent.

Here is a descriptive specification for the *agent_responds* transition for simple, non-terminal calling agents:

```
∀ system_state.  agent_responds system_state =
  If no messages in internal queue then
    return system_state
  else begin
    read message from the head of the internal queue;
    let src be the ID of the message sender;
    if src is the terminal agent then
      do send_response
    else
      do save_request
  end
```

As in the *agent_responds* transition for terminal agents, if there is no message in the mailbox then the transition implements a stuttering step.

When a message is from the terminal agent ($B$ in our example), it is a response message to a previous request message from the non-terminal agent ($A$). The message from $B$ corresponds to the request message at the head of $A$'s internal buffer; $A$ buffers the request messages in the order that it receives them and $B$ responds to $A$'s requests in the order that they are received. $A$ uses the value in $B$'s response message to calculate its response message to the sender of the original request message. The specification for this action is shown here:

```
∀ system_state. send_response system_state =
    get original request message from head of internal buffer;
    let rsrc be the identity of the original request message sender;
    let bdata be the data value returned from the terminal agent;
    create response message that contains bdata;
    put the response message in rsrc's mailbox;
```

If the message is not from the terminal agent — i.e., the message is a new request message — the message is added to the end of the internal queue and $A$ sends a request message to $B$.

```
∀ system_state. save_request system_state =
    save request message at end of buffer;
    let data be the data value passed in the request message;
    apply function a, giving adata;
    create request message that contains adata to send to terminal agent;
    put the request message in terminal agent's mailbox;
```

#### 5.4.2.2 Composing a Chain of Calling Agents

When an agent sends a request message to $A$, which appears to $A$ as an *env_send* transition, the transition appears to $B$ as an *env_arbitrary* transition by the environment (because $B$'s mailbox is unchanged). Similarly, when $A$ performs an *agent_reads* transition, it appears to $B$ as an *env_arbitrary* transition.

When $B$ gets a request message from $A$ and performs an *agent_reads* transition, it appears to $A$ as an *env_arbitrary* transition. When $B$ sends a response message to $A$, that transition appears to $A$ as an *env_send* transition.

$A$'s *agent_responds* transition appears to $B$ as either an *env_send* or *env_arbitrary* transition, depending on whether $A$ is performing a *save_request* ($A$ sends a request message to $B$) or *send_response* ($A$ sends a response message to $E$) action, respectively.

#### 5.4.2.3 Refinement Step

In chapter 7 we will give a detailed example of the refinement of a chain of calling agents like $A$ and $B$. Here we will give a high-level overview to show how the refinement of this system differs from the refinement of the terminal calling agents.

The refinement step must show that the composition of $A$ and $B$ satisfies the specification for $AB$. $AB$ is a terminal agent, so its specification is similar to that of $B$, but it returns the function value $ab(x)$ in its response messages. One important proof obligation is to show that $a(b(x)) = ab(x)$.

All request messages from outside the system that contains agents $A$ and $B$ are sent to $A$'s mailbox. For this reason, our refinement mapping must map $A$'s mailbox to the mailbox for $AB$. Because we assume here that $B$ only receives messages from $A$, $B$'s mailbox becomes state that is internal to the system, so $B$'s mailbox is removed from the global state by the refinement mapping.

The terminal agent $B$ sends messages to $A$, and these messages appear in $A$'s mailbox (and internal queue). In the specification for $AB$, however, there are no state transitions that could account for the appearance in $AB$'s mailbox of the messages sent by $B$. For this reason, our refinement mapping hides the messages that are sent to $A$ by $B$, leaving all other messages and their ordering alone.

The mapping of $A$'s internal queue and buffer to $AB$'s internal queue is simple. $A$ saves each request message in its buffer until it receives a response message from $B$. The concatenation of $A$'s queue (without the messages from $B$ that are hidden by the refinement function) with $A$'s buffer contains the same messages in the same order as the request messages in $AB$'s internal queue. $B$'s internal state is completely filtered by the mapping.

This mapping is depicted in figure 5.11, showing how messages from $B$ are filtered by the refinement function and how $A$'s queue and buffer are concatenated to form $AB$'s queue.

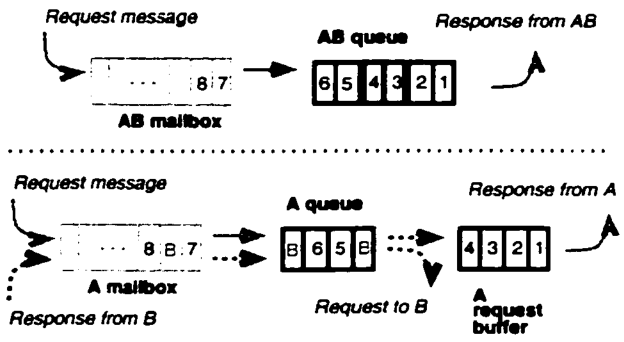

Figure 5.11: Mapping of $A$ and $B$ state to $AB$ state

#### 5.4.2.4 Mapping the Transitions

$B$'s mailbox is hidden by the refinement, so when $B$ performs an *agent_reads* transition, no change in $AB$'s state is observable. Thus, all of $B$'s *agent_reads* transitions map to stuttering steps in $AB$. Similarly, whenever $B$ sends a response message to $A$, that message is filtered by the refinement function so that there is no change to $AB$'s mailbox, so all of the *agent_responds* transitions performed by $B$ also map to stuttering steps in $AB$.

When $A$ does an *agent_reads* transition, but reads a message from $B$, no change to $AB$'s external or internal state is visible (because the refinement function filters out all the messages from $B$), so this transition maps to a stuttering step in $AB$. If, however, the message read by $A$ is not from $B$, the transition maps to an *agent_reads* transition in $AB$.

When $A$ sends a request message to $B$, the changes to $B$'s mailbox and to $A$'s queue are filtered by the refinement function, so the transition maps to a stuttering step in $AB$. When $A$ sends a response message after receiving a response from $B$, that transition maps to an *agent_responds* transition by $AB$.

## 5.5 Adding Fault Tolerance to the Calling Agent Model

Our calling agent model implicitly assumes that there will always be one response message for every request message (e.g., in the previous section, $A$ implicitly assumes that $B$ will send only one response to every request sent by $A$). In fact, as we will show in chapter 7, we can explicitly prove that a system of calling agents has this property.

But real systems may not necessarily always conform to this assumption. Our model of a calling agent can be made more sophisticated to handle "spontaneous" or otherwise erroneous response messages. This can be handled in two different ways:

1. The environment assumption can be strengthened to exclude the sending of erroneous response messages. The behavior of a calling agent is guaranteed provided that the environment satisfies the agent's environment assumptions. By strengthening the environment assumptions to exclude erroneous response messages, the behavior of a calling agent if it receives an erroneous response message is left undefined.

2. The calling agent state transition specifications can be strengthened to explicitly describe the behavior of the calling agent should it receive erroneous response messages.

In general, modifications to the basic calling agent model to handle any kind of faults can be implemented in the same two ways as just described for erroneous response messages. There are advantages and disadvantages to each approach. Strengthening the environment assumption will most often result in a simpler specification than strengthening the state transition specifications. On the other hand, the behavior of calling agents in the presence of faults (i.e., when the environment does not satisfy some properties that are considered to be "normal") might be an important feature that must be captured in the specification.

## 5.6 Summary

In this chapter, we have described our calling agent model and have introduced how the composition method can be applied to a system of calling agents. Calling agents can be characterized by whether or not they maintain part of the functional system state, and whether or not they call other agents. Agents that do not call other agents are known as *terminal calling agents.*

The composition of terminal calling agents is relatively simple because there is no interaction between the agents. The composition of non-terminal agents is a much more complicated proof. In this chapter, we gave an overview of the composition of a simple chain of agents. In the next chapter, we will explain how the composition method can be applied to more general topologies of calling agents called *calling agent hierarchies*, and in chapter 7 we will give a fully worked example of the compositional proof of a system of calling agents.

# Chapter 6

# Templates and Calling Agent Hierarchies

In aught that tries the heart, how few withstand the proof!
*Byron - "Childe Harold," Canto II, Stanza 66*

When a system consists of a large number of components, even compositional proof can be quite complex. Our method of reducing the complexity of compositional proof hinges on being able to incrementally compose systems that contain a large number of components and on being able to reuse specifications and proofs for each incremental proof stage.

Incremental composition reduces the number of components that are composed at a time. The result of composing a set of components into a subsystem is that the details of the interaction between those components become internal to the composed subsystem and, hence, hidden when composing the subsystem with other components. The greatest benefit in terms of reducing compositional proof complexity will occur when each incremental stage hides a maximum of these details.

While using incremental composition stages can reduce the complexity of each composition stage, a large number of stages will also increase the complexity of a proof. An optimal incremental proof strategy, therefore, is one that only uses an incremental stage when doing so will hide detail and that chooses the components to be incrementally composed that will result in hiding the most detail possible. Our method of finding an optimal incremental proof strategy is based on viewing a system of calling agents as a hierarchy and by incrementally composing from the bottom of the hierarchy toward the top.

Viewing a system of calling agents as a calling agent hierarchy can optimize the number of incremental proof stages, but there may still be a large number of stages. Ideally, specifications and proofs can be reused in each incremental stage to reduce the work in each stage. In principle, in a system that has a sufficiently regular, recursive topology, a single set of reusable specifications and one reusable compositional proof could suffice to verify the entire system.

In order for reuse of specifications and proofs to be practical, we need to have abstract specifications and generic proofs that can be instantiated to support a wide variety of calling agent types. We call the abstract specifications and proofs *templates*. Templates are parameterized, so an instance of a calling agent can be specified and composed with other agents simply by providing appropriate arguments.

In this chapter, we describe the characteristics of a calling agent hierarchy and an algorithm for finding an optimal incremental composition strategy to compose the hierarchy. We also explain how templates can be applied to a hierarchy to incrementally compose a large system of calling agents.

## 6.1 Templates

A template includes both abstract specifications and generic proofs. What makes a specification abstract and a proof generic is the use of parameters. The parameters can be values, types, specifications, and theorems. A template can be saved in HOL as a *generic theory* [38].

A generic theory in HOL is a collection of definitions and theorems that provide a framework for theorem reuse. It consists of an abstract representation of objects and types, abstract operations, and theorems about the representation. A generic theory also includes a set of theory obligations. The abstract theorems are guaranteed to hold for any instantiation of the abstract representation provided that the theory obligations are satisfied.

### 6.1.1 Parameterized Specifications

HOL provides a "generic typing" feature that can be used to parameterize types. For example, a generic function $f$ whose domain and range may be different types can be defined using the type definition $f : (* \rightarrow **)$, which means that the domain of $f$ is of type $*$ and the range is of type $**$.

Higher level constructs, such as functions, can themselves become parameters by using universal quantification. For example, we can change the following descriptive specification from section 5.4.1 for the *agent_responds* transition

```
∀ system_state. agent_responds system_state =
  If no messages in internal queue then
    return system_state
  else begin
    return the system state modified as follows:
    read one message from the head of the internal queue;
    let src be the ID of the message sender;
    let data be the request data value;
    put a response message in src's mailbox that contains f(data)
  end
```

into a more abstract specification by universally quantifying the function $f$ that the transition calculates.

```
∀ system_state (f : (* → **)). agent_responds system_state =
  If no messages in internal queue then
    return system_state
  else begin
    return the system state modified as follows:
    read one message from the head of the internal queue;
    let src be the ID of the message sender;
    let data be the request data value;
    put a response message in src's mailbox that contains f(data)
  end
```

This parameterized specification is generic for all transitions that apply a single function to the data value in the request message.

### 6.1.2 Generic Proofs

A generic proof is parameterized by the use of the abstract specifications and by a set of theory obligations. For example, as described in section 5.4.2.3, when composing a simple chain of calling agents $A$ and $B$ into the system $AB$, it is necessary to show that the composition of the functions calculated by $A$ and $B$, $a(b(x))$, is equal to the function calculated by $AB$, $ab(x)$. A template can be created to compose the agents $A$ and $B$, and any other pair of agents that are structurally homogeneous to $A$ and $B$, by universally quantifying the functions, with a corresponding theory obligation to show that $a(b(x)) = ab(x)$.

The same template can be applied to compose an arbitrarily long chain of structurally homogeneous calling agents. For example, consider a chain of calling agents shown in figure 6.1 that consists of $A$, which calls $B$, which calls $C$, which calls $D$. To compose this chain, we would first apply the template to compose $C$ and $D$ into $CD$, then apply the template to compose $B$ with $CD$ into $BCD$, and then apply the template one last time to compose $A$ with $BCD$. At each incremental step in the composition, however, in order to be able to apply the template we need to satisfy the theory obligation, instantiated with the functions from each calling agent (or composition of calling agents).

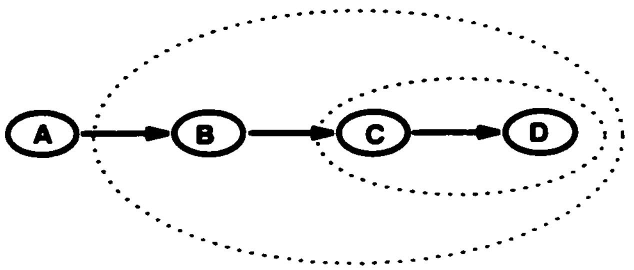

Figure 6.1: Incremental composition of a "long" chain of calling agents

## 6.2 Calling Agent Hierarchies

The abstraction of a calling agent hierarchy is a method of partitioning a system of calling agents so that an optimal incremental composition strategy can be found. An optimal incremental composition strategy composes components that interact, in order to hide the details of that interaction in later incremental composition stages. As explained in section 5.4.1.2, composing two terminal agents does not hide any detail. In fact, composing any set of agents that do not interact hides no detail. An optimal incremental composition strategy will, whenever possible, only compose agents that interact.

The system depicted in figure 6.1 is a simple example of a calling agent hierarchy. The incremental proof strategy described in section 6.1.2 to compose that system is optimal in the sense that it only composes agents that interact. There would be no benefit, for example, in having an incremental composition stage that composed agents $B$ and $D$.

In this section, we will discuss more complicated templates and calling agent hierarchies. Although we focus here on hierarchies whose call graphs are acyclical, our techniques can be applied to systems whose call graphs contain cycles as well. The chief difference between templates for acyclical and cyclical call graphs is that templates for cyclical graphs must be more sophisticated with respect to the eventual generation of responses (i.e., they must show that the cycle eventually terminates).

### 6.2.1 Composing n-ary Trees

A linear chain of calling agents is a unary tree. Templates can be developed in a similar manner for binary and larger order trees. For example, consider a system of calling agents whose calling structure is a complete binary tree, as shown in figure 6.2. Each of the subtrees — $(B, 1, 2)$ and $(C, 3, 4)$ — that consist of a root node and two leaves can be composed, leaving another instance of a root node ($A$) and two leaves ($B12$ and $C34$) that can be composed using the same template.

#### 6.2.1.1 Identity Agents

In the case where an $n$-ary tree is incomplete, so that some of the subtrees have a lower order than others, we do not necessarily have to use different templates for each order

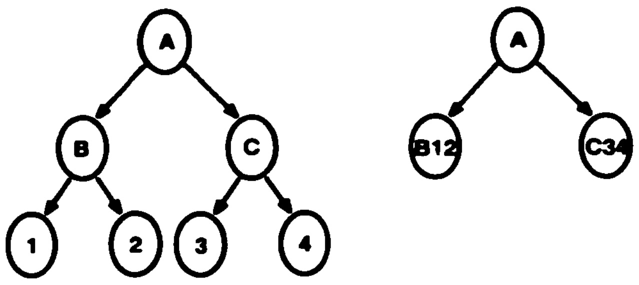

Figure 6.2: Binary tree calling agent hierarchy

subtree. As an alternative, a template for an $n$-ary subtree can be used to compose $m$-ary subtrees ($m < n$) by using an *identity agent*. An identity agent is a simple terminal agent that is defined to have no effect on the system state other than to receive request messages and send responses. The function of an identity agent depends on the requirements of a template. For example, in a template where agents are passed parameters and return function values, an identity agent might simply return the input value unchanged (i.e., it implements the identity function).

For example, as depicted in figure 6.3, agent $C$ only calls agent $D$, but in order to be able to use the same template for the entire hierarchy $C$ is specified to also call an identity agent $i$.

An $m$-ary agent can call $n - m$ identity agents to "fill" the unused calls in the template. Although the specification for the $m$-ary agent is no longer $m$-ary, but is now $n$-ary, the specifications are functionally equivalent because the identify agents have no effect.

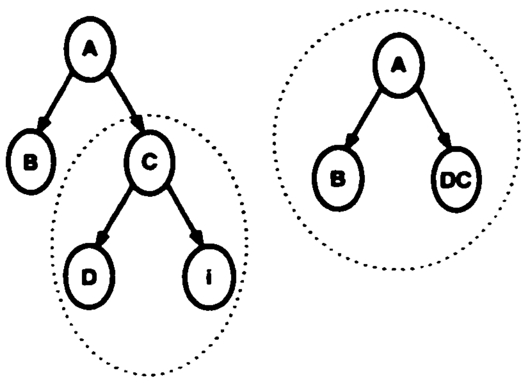

Figure 6.3: Composition of an incomplete tree

### 6.2.2 Conditional Call Graphs

Up until now, we have only been considering "hard-coded" call graphs. That is, if an agent ever calls another, it always calls that other agent in every instance. For example, consider the binary call graph depicted on the left side of figure 6.3, above. Agent $A$ might calculate a function of the values returned by both agent $B$ and agent $C$. In this case, both agents $B$ and $C$ would always be called. We can describe $A$'s actions with the statements

$$B(x); C(x)$$

to indicate that $A$ calls both $B$ and $C$ using the input parameter $x$. In this section, we consider calling agents that may call a subset of all the agents that they could possibly call, depending on different parameter or system state values.

#### 6.2.2.1 Parallel Conditional Call Graphs

The agent $A$ described above, which always calls agents $B$ and $C$, implements what we define as an *unconditional parallel call.* The call is unconditional because both $B$ and $C$ are always called. The call is parallel because the order that results from $B$ and $C$

are returned to $A$ does not matter, so $A$ can call $C$ without waiting for a response from $B$. ($A$ must wait for responses from both $B$ and $C$, however, before it can return a response of its own.)

Consider, however, an agent $A$ that implements the following conditional statement based on an input parameter $x$:

$$if\ x > 5\ then\ B(x)\ else\ C(x).$$

This case is an example of a *conditional parallel call.* We can use the same template for a conditional parallel call as we do for an unconditional parallel call. Consider an agent $A$ that implements the following conditional statements:

$$if\ f(x)\ then\ B(x)\ ; if\ g(x)\ then\ C(x).$$

If $f$ and $g$ are both trivially TRUE, the function will always call both of the other agents and we have an unconditional parallel call. If $g = \neg f$, the function implements a conditional where only one of the branches is called. Thus, different variations on parallel calls can be implemented by using different instantiations of $f$ and $g$.

#### 6.2.2.2 Sequential Conditional Call Graphs

Not all binary call trees, however, describe parallel calls. The call to $C$ may depend on a value returned by $B$, so that $A$ cannot call $C$ until $A$ receives $B$'s response. For example, $A$ may pass to $C$ the value that was returned by $B$. This dependency leads to an *unconditional sequential call*, which we can represent by the statement $B$ *then* $C$.

The value returned by $B$ may determine if $A$ calls $C$ at all, which could be represented by the conditional $if\ f(B)\ then\ C$. If $f$ is trivially TRUE then the conditional implements an unconditional sequential call. Thus, we can use a single template for both unconditional and conditional sequential calls by instantiating with the appropriate function $f$.

### 6.2.3 Composing Forests of Calling Agents

As described in section 5.4.1.2, the refinement function for composing two independent, terminal calling agents is essentially the identity function. This fact holds true in general for composing any forest of calling agents. Because the agents in the forest do not call each other, all of their state and state transitions are preserved by the composition and refinement. Composing a forest of agents is the only time that an optimal incremental composition strategy should compose non-interacting agents.

To compose a forest of call trees, we first compose each of the trees individually and then compose the resulting composed agents as non-interacting terminal agents. This is depicted in figure 6.4, where first $A$ and $B$ are composed, $C$ and $D$ and composed, then $AB$ and $CD$ are composed.

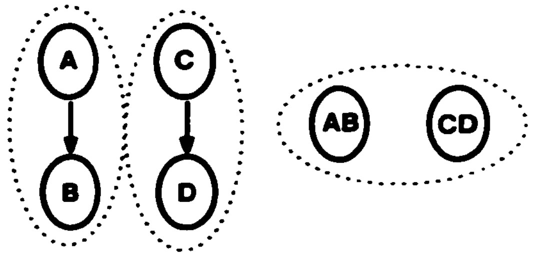

Figure 6.4: Composition of a forest of calling agents

#### 6.2.3.1 Compound Agents

The result of the composition of the two independent trees in figure 6.4 is a *compound* agent, which has more than one entry point. Each entry point is a distinct destination to which a request message can be sent.

The *degree of a compound agent* is the number of entry points that the compound agent has. For completeness, we refer to an agent with a single entry point as a compound entry point with degree 1.

As with incomplete $n$-ary tree call graphs, where some of the nodes have a degree that is less than $n$, some agents in a system may be compound agents of degree 1, others of degree 2, and others with even higher degrees. To write templates that can handle agents of different degree, we can make every agent a compound agent of some maximal degree $n$. For agents of degree $m$, where $m < n$, the extra $n - m$ entry points can be assigned to identity agents.

## 6.3 Composing with Reuse

The examples that we have given until this point do not "reuse" agents, i.e., no agent is called by more than one other agent. When an agent is called by more than one other agent, however, the resulting call graph forms an acyclic directed graph that is not a tree. An example of this is shown in figure 6.5, where agent $C$ is called by both agents $A$ and $B$.

The method of recursively composing trees that we have previously described does not work when agents are reused. The problem is that once two agents are composed they

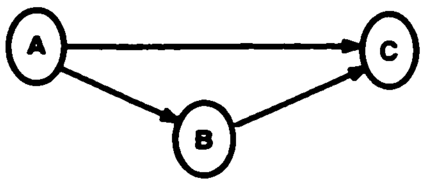

Figure 6.5: Directed acyclic graph of calling agents that is not a tree

lose their individual characteristics and become a single, composed entity.

The solution is to compose agents so that the entry point of the called agent becomes part of the composed system. The resulting system is a compound agent, as if two independent trees had been composed. For example, to compose the system depicted in figure 6.5, we could use a binary tree template to first compose agents $B$ and $C$ into the compound agent $\{BC, C\}$ (using an identity agent for $B$'s other call), propagating the entry point for $C$. The next step is to compose the compound agent with agent $A$. In chapter 7, we give an example template that composes a system like $B$ and $C$ into a compound agent.

### 6.3.1 An Algorithm for Composition with Reuse

Assuming that we have suitable templates, we can compose arbitrary rooted, acyclic call graphs in the manner just described — composing in a depth-first manner and propagating entry points — using the following algorithm:

We start with a set $A$ of agent entry points, $\{a_0, a_1, ..., a_n\}$, where the calling topology forms a rooted, acyclic graph. We also have a corresponding set $E$ of the degree of incoming edges to each entry point in the graph, $\{e_0, e_1, ..., e_n\}$. Entry points in the roots of the graph are the entry points into the system as a whole, and their edge count is always 1.

Composition is done iteratively. In each iteration, do the following:

1. Calculate the longest path in the graph (if there is more than one longest path, pick any one of them), then compose the agents in the smallest subtree that contains the farthest edge. For example, in figure 6.5 the longest path is $(A, B, C)$ and the smallest subtree that contains the farthest edge is $\{B, C, i\}$ (where $i$ is an identity agent). In figure 6.2 the longest paths are $(A, B, 1)$, $(A, B, 2)$, $(A, C, 3)$, and $(A, C, 4)$. We can choose as the smallest subtree that contains the farthest edge either $\{B, 1, 2\}$ or $\{C, 3, 4\}$.

2. For each entry point $a_i$ that is in, but is not the root of, the subgraph that is being composed, subtract one from $e_i$. Any entry point whose incoming edge count is reduced to zero has been fully composed, so it no longer needs to be propagated (i.e., it becomes an internal, hidden, detail). If, however, after the composition, an entry point in the subtree has a non-zero incoming edge count, then it must be propagated.

   For example, when composing the system in figure 6.5, the incoming edge count for $C$ is initially 2. After it has been composed with $B$, however, its edge count is decremented to 1. $B$'s edge count is not decremented because it is the root of the composed subtree. Because both $B$ and $C$ have non-zero edge counts, their entry points are both propagated.

   In the system in figure 6.2, however, once the subtree $\{C, 3, 4\}$ has been composed, the edge counts for entry points 3 and 4 are decremented to zero, so those entry points are not propagated. $C$'s edge count remains at 1 because it is the root of the subtree, so it is propagated.

3. Repeat the steps above until the only entry points left in the system are root entry points (i.e., the edge counts for all remaining (propagated) entry points and the longest paths are all 1). Then compose the root entry points as terminal agents.

For example, consider the call graph depicted in figure 6.6. The incoming edge counts for agents $\{A, B, C, D\}$ are $\{1, 1, 2, 2\}$, respectively. Initially, the longest path in the tree is $(A, B, C, D)$.

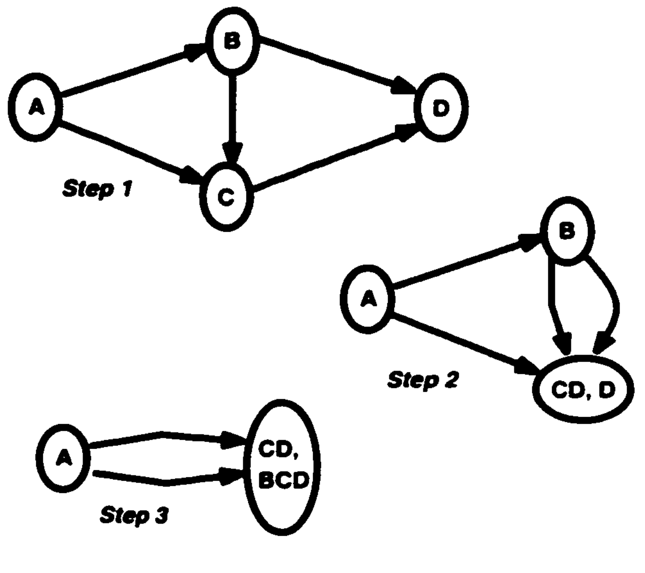

Figure 6.6: Example of composition with reuse

In the first step, we compose $C$ and $D$ (and an identity agent). Both $C$ and $D$ initially have incoming edge counts of two, but the incoming edge count for $D$ is decremented because of the composition with $C$. Because $D$'s incoming edge count is still greater than zero, however, we must propagate its entry point. The result of the composition is a compound agent with two entry points: $CD$ and $D$. The new set of entry points is now $\{A, B, (CD, D)\}$, and the corresponding set of incoming edges is now $\{1, 1, (2, 1)\}$.

In the next step, the longest path in the tree is $(A, B, (CD, D))$. We compose $B$

with $(CD, D)$. The incoming edge count of $D$ reduces to zero, so that entry point is no longer propagated. The incoming edge count of $CD$ is reduced to one. As a result of this step, the set of entry points is reduced to $\{A, (BCD, CD)\}$, and the incoming edge counts are $\{1, (1, 1)\}$. The final composition step fully composes the system, leaving us with the entry point $\{ABCD\}$ and an edge count set of $\{1\}$.

### 6.3.2 Reuse with Multiple Roots

This method of composition also works for directed, acyclic call graphs with multiple roots. For example, consider the call graph in figure 6.7.

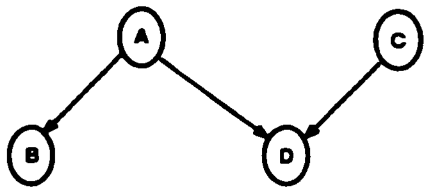

Figure 6.7: Acyclic system of calling agents with multiple roots

Initially, the set of entry points is $\{A, B, C, D\}$, and the corresponding set of incoming edge counts is $\{1, 1, 1, 2\}$. The three longest paths are the same length, so we can compose either of the subtrees $\{A, B, D\}$ or $\{C, D\}$ (with an identity agent) first. Let us choose $\{C, D\}$, so the result of the first composition is the agent set $\{A, B, (CD, D)\}$, and the incoming edge count set $\{1, 1, (1, 1)\}$.

In the next step, the longest path is in the subtree rooted at $A$, so we compose $A$ with $B$ and $D$. The edge count for $CD$ remains greater than 0 so that entry point is propagated. The result is the compound agent $(ABD, CD)$, an edge count set of $\{(1, 1)\}$, and we are done.

### 6.3.3 Correctness Proof of Composition with Reuse Algorithm

To prove that this algorithm works for all acyclical call graphs, we first assume that we already have templates for all compositions. We must also define what it means for the algorithm to fail. What is a wrong result? The algorithm fails if either

1. it attempts to compose an agent with another agent that is not completely composed, or
2. the result of a composition calculates the wrong function.

The first error case is impossible because if we try to compose an agent with some agent that is not already completely composed, the agent that is not already composed must itself be calling yet another agent. If so, then there is still a longer path that has not been composed, but this violates the rules of our algorithm, which always composes the end of the longest path first.

The second case is also impossible because we use the call graph to determine what to compose and the templates guarantee that the compositions are correct.

The algorithm for composition with reuse described above always yields an optimal incremental proof strategy — it only composes agents that interact except for the final stage, when there is nothing left but root entry points. This is guaranteed because the algorithm at each step composes the smallest subtree that contains the farthest edge. An edge is only present when one agent calls another, so the agents that are composed must interact.

## 6.4 Servers with State

Creating templates for servers that maintain system state is trickier than for stateless servers. By "stateless" we mean that the server does not maintain any global state beyond its mailbox. The trouble is that the refinement step must be able to refine the state and the transitions but, in general, the state maintained by servers with state will vary considerably in terms of its type and the operations that may be performed on it. This kind of structural non-homogeneity may not be as amenable to the use of templates. Where system state is structurally homogeneous, however, templates are quite useful.

When the global state is structurally homogeneous at each stage of composition, templates can be used in the same way as for the purely stateless servers described earlier in this chapter. Once the compositional proof has been done abstractly, it can serve as a template for composition of the entire system.

For example, consider the distributed name service (DNS) type system described in section 2.1.2. When name servers make recursive queries, it appears to the original caller that the called name server itself knows the IP addresses of every system in its domain. In other words, the called name server and all the name servers beneath it can be composed into a single, comprehensive name server that implements a mapping from every host name in the domain to corresponding network (IP) addresses.

We can represent the mapping from host names to IP addresses by each name server in the form of a table — a list of host names and their corresponding IP addresses. The global state for the composition of the entire system would be a single, very large table, and the refinement function merges the smaller tables of each name server into the larger table. This is depicted in figure 6.8.

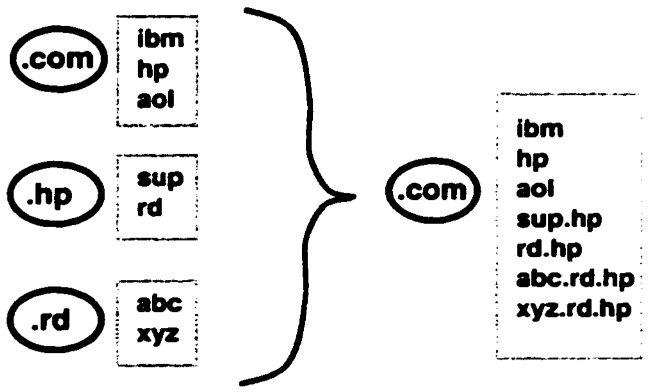

Figure 6.8: Composition of global state in DNS-type system

At each step of the composition, the state of each of the components — their host name to IP mapping tables — are mapped to the larger table of the composed system. When state is updated, the update is reflected in the mapping. The structure of the address tables is the same at every step of the composition and, more importantly, the refinement mapping is the same. For this reason, the same composition template can be applied when composing each of the name servers with the rest of the system.

### 6.4.1 Example: A Micro-kernel Operating System

As another example of how our method can be applied to compose a system of servers that have regular state, consider a simple micro-kernel operating system. The system consists of a number of server tasks, including a file manager, memory manager, and I/O servers. To simplify our discussion, here we will consider just the operating system calls that create user processes and that manage files. These services are provided by the memory and file managers and by a low-level "system" task whose job it is to manage the kernel process table. This design is similar to that used in the Minix operating system [37]. For simplicity, we will assume that the abstractions of messages and mailboxes, and of process scheduling, are provided by the kernel.

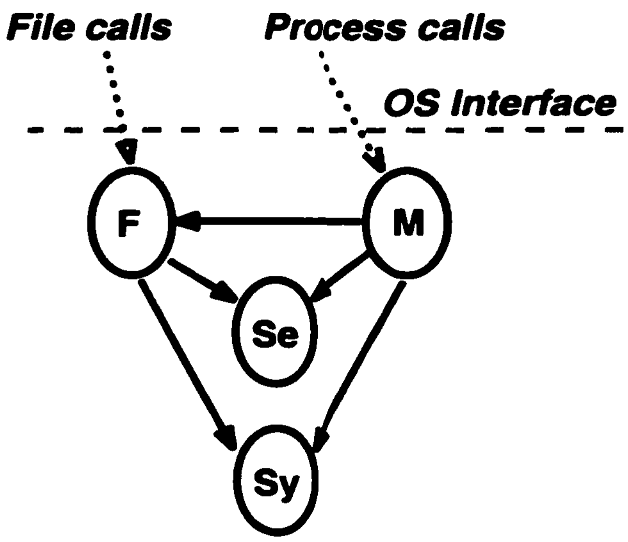

Figure 6.9: Simple operating system

In addition to the file manager, memory manager, and system task, our system also includes a "security" server that implements a mandatory access control policy. We model each of the server tasks as a calling agent. The calling agent hierarchy for this system is depicted in figure 6.9. As shown in the figure, user processes invoke the file manager ($F$ in the figure) for file management operations (open, close, etc.) and the memory manager ($M$ in the figure) for process management operations (fork, exec, etc.). The memory manager calls the system task ($Sy$), the security server ($Se$), and the file manager. The file manager calls the security server and system task. (The file manager must also call a disk I/O server, but for purposes of this discussion we will assume that the functionality provided by the disk I/O server is already present in the file manager.)

All calls from user processes to server tasks, and between server tasks, are implemented as messages. Because each server task has only one mailbox, different types of calls to the same server task, for example the *fork* and *exec* calls to the memory manager, are differentiated by a *call id* parameter in each request message.

### 6.4.1.1 Server Tasks

The system task manages the kernel process table, which has one record per process. A process's record in the process table contains the process's registers (when the process is suspended), its memory map (the locations and boundaries of the process's stack, data, and code segments), and other low-level data. The memory manager calls the system task when a new process is created (via a *fork* call) to create a record for that process in the process table and to create a mailbox for the process. The file manager calls the system task when a new process is created and when an existing process begins to execute a new code segment (via the *exec* call) to establish the new instruction pointer and code segment size.

The security server maintains a table of user identity and clearance information for each process. The file manager consults the security server to determine access permissions. For example, if a process wants to open a file for read access, the file manager consults the security server, which compares the process's user clearance against the file's classification and allows or disallows access. The memory manager calls the security server when a new process is created to create a record for that process in the identity/clearance table and when a process changes its effective user id.

The file manager maintains a table of file information for each process. The per process information includes open files, access modes, and file pointers. Calls from user processes to open and close files, and to read and write files, go directly to the file manager. The memory manager calls the file manager when a new process is created to create a record for that process in the file table. The memory manager also calls the file manager to load an executable program file for a new process and when an existing process *execs* a new program. In both of these cases, the file manager calls the security server to see if the process has execute access permission to the specified program files and calls the system task to update the code segment information for the process.

The memory manager is really a process manager. It assigns each process a block of memory when the process is created, and reclaims memory when a process exits or is killed. Calls to *fork* (create) a new process, or to *exec* a new program go directly to the memory manager. When a new process is created, the memory manager assigns it a memory block and creates a record for the process in the memory manager table. Among other information, the memory manager table contains information about the process's parent that can be used when a process exits or is killed.

After creating a memory manager record for a new process, the memory manager calls the system task to create a process table record and mailbox for the new process, calls the security server to create an identity/clearance record for the process, and calls the file manager to create a file table record for the process. When a new process is created and when an existing process *execs* a new program, the memory manager calls the file manager to load a program file for the process to execute.

### 6.4.1.2 Composition of the Server Tasks

Each of the server tasks keeps a table of information that is organized on a per-process basis. A process can be completely described by the concatenation of the appropriate records from each of the tables. The concatenation of the tables maintained by the server tasks is a refinement of the operating system specification with respect to processes and the files that they have open. This is depicted in figure 6.10. As shown in the figure, the refinement function is applied to the state maintained by each server task to produce the state maintained by the composed system.

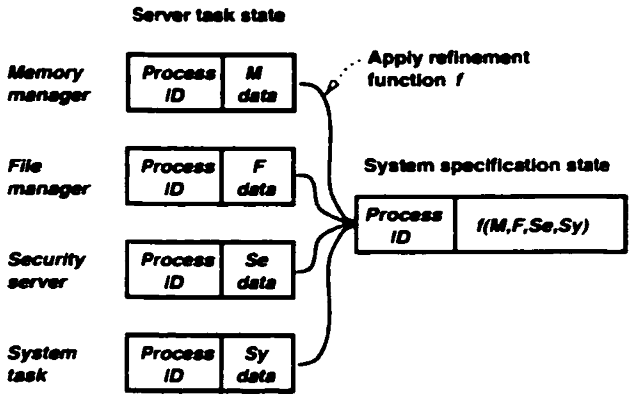

Figure 6.10: Concatenation of operating system tables

To compose this system, we can use a single template in two incremental composition stages. The template composes a server task and the server tasks that it calls. Following our algorithm for an optimal incremental composition strategy, we first compose the subtree that contains the farthest edge. That subtree is rooted at the file manager (see figure 6.9). Applying the template, we compose the file manager with the security server and system task. Because the edge counts for the security server and system task are still greater than zero after this step, we propagate their interfaces.

In the second composition stage, we compose the memory manager with the file manager, security server, and system task. After this step, we are left with a composed system that presents the file manager and memory manager interfaces, and our composition is done.

## 6.5 Summary

In this chapter, we have described how to view a system of calling agents as a calling agent hierarchy, and an algorithm for determining an optimal incremental composition strategy in a calling agent hierarchy. We have also described templates — collections of abstract, parameterized specifications and generic proofs — that can be reused in the composition of a structurally homogeneous calling agent hierarchy, and presented some examples of how templates can be applied to compose systems. In the next chapter, we will give a fully worked example of a template, including parameterized specifications and a generic proof.

# Chapter 7

# An Example Template

We'll leave a proof by that which we do.
*Merry Wives of Windsor, Act IV. Scene 2*

In this chapter, we present an example of our generic compositional proof methodology, developed using HOL. The example template consists of two abstract calling agent specifications: a specification for a terminal agent (as described in section 5.4.1.1), called $F$, and a specification for a non-terminal agent, $FG$, that calls the terminal agent (as described in section 5.4.2). The example template composes the agents into a compound system, so that the interfaces to both agents are propagated (i.e., this is an example of "composition with reuse," as described in section 6.3). We will refer to the interfaces in the composed system as the *OSF* and *OSFG entry points* to distinguish them from the interfaces to the calling agents.[1]

The proof of this template required approximately 23,000 lines of HOL. Roughly 1,600 lines (about 7%) were required for the composition step. Of the remainder, refinement

[1]In this context, "OS" can be considered to mean "overall system."

of the progress properties took three times more work than the refinement of the safety properties.

This chapter is organized as follows: Section 1 gives a brief architectural overview of the template. Section 2 explains how we specify traces, including the global and local state. Section 3 explains the entry point specifications, while section 4 explains the calling agent specifications. Section 5 covers the environment specifications. Section 6 discusses the composition step of the proof. Section 7 discusses the refinement step for safety properties, section 8 explains how the proof obligations are used in the refinement step, and section 9 explains the mapping of the progress properties in the refinement step. Sections 10 and 11 explain how the template can be applied and how the example template can be further generalized.

## 7.1 System Overview

The system is depicted in figure 7.1. Agent $F$ accepts request messages containing some value $y$ (of HOL numeric type *num*) and returns the value $f(y)$ (where $f$ is a function of type $num \rightarrow num$). Agent $FG$ accepts request messages that contain a value $x$ (also of type *num*), sends the value $g(x)$ ($g$ is a function of type $num \rightarrow num$) in a request message to agent $F$, which applies function $f$ to the request message value and returns $f(g(x))$ to $FG$. $FG$ returns the result from $F$ in its response message.

The specifications for $F$ and $FG$ are parameterized (see section 6.1.1) by the functions $f$ and $g$, respectively. The template can be used to compose any agents that are structurally homogeneous with $F$ and $FG$ by instantiating different functions $f$ and $g$.

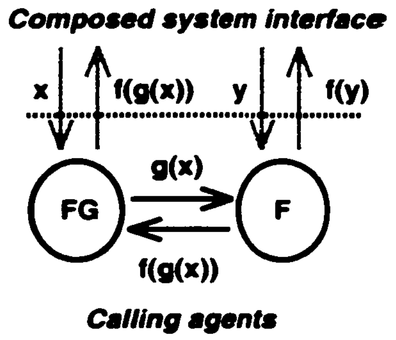

Figure 7.1: Entry points and calling agents

The composed, compound system has two entry points: *OSF* accepts requests that contain $y$ and returns $f(y)$, while *OSFG* accepts request messages that contain a value $x$ and responds with the value $fg(x))$ (where $fg$ is a function of type $num \rightarrow num$). The refinement step in the composition proof shows that $f(g(x)) = fg(x)$.

### 7.1.1 Basic Form of the Specifications

Each of the composed system entry points and the two calling agents have the same basic state transitions: to read a message from a mailbox and to send a response message. This basic form is depicted in figure 7.2. Variations on this basic form are due to whether a specification applies to an entry point or to a calling agent, and whether or not a calling agent is a terminal agent. Each state transition has a corresponding progress property that specifies that the transition happens infinitely often.

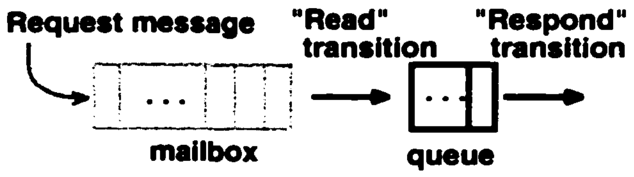

Figure 7.2: General form of specifications for example template

## 7.2 Trace Specifications

As described in section 4.3, we define traces to be functions from natural numbers to the triple $\langle agent, (globalstate, internalstate)\rangle$. Here we give the precise HOL trace definitions for both the agents and for the composed system, and introduce the constructors and destructors that are used to manipulate the state objects.

### 7.2.1 Global State

The global state for both the composed system and for the two agents is the set of mailboxes. The *mboxes* type is defined as a set of cartesian product type

$$(mbox\_id\#mbox\_def)$$

Every mailbox has a unique identifier, *mbox_id*, that matches the identifier of an agent and that designates the mailbox as belonging to that agent. The *mboxes* set effectively defines a partial function, or mapping, from identifiers to mailboxes.

The contents of a mailbox are defined by the type *mbox_def*. Mailboxes are defined as lists of messages. New messages are intended to be added by the system to the end of the list (we say "intended" here because the type definition cannot guarantee that the state is used in this manner). Messages are intended to be read by an agent from the head of the list. A *next* pointer (i.e., a numerical index into the list of messages) is used to keep track of the next message that is to be read. The *mbox_def* type, therefore, is defined as

$$(next\#((msg\_def)list))$$

where messages are of type *msg_def*. The structure of the mailboxes is depicted in figure 7.3.

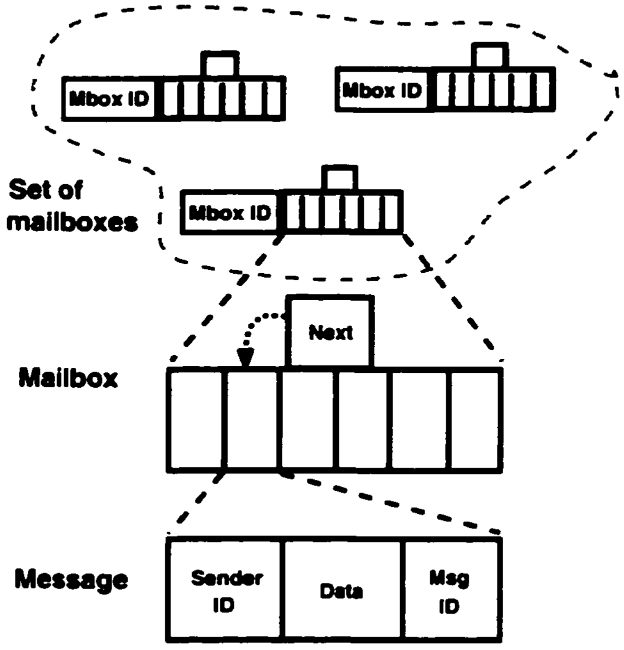

Figure 7.3: Structure of the mailboxes

This definition of mailboxes, using the "next" pointer, was inherited from a previous project. An alternative specification that would be much easier to formally reason about would be a simple queue, where messages are appended to the queue as they arrive and removed from the queue as they are read.

The *msg_def* type is a triplet: $(task\_id\#msg\_data\#msg\_id)$. The *task_id* is the identifier of the agent that sent the message. The *msg_data* is the actual data contained in the message. In the example in this chapter, *msg_data* is the HOL numerical type, *num*.

The *msg_id* is an anachronism and is not used.

### 7.2.2 Constructors and Destructors

Various constructors and destructors are defined to manipulate messages, mailboxes, and the mailbox set. Destructors *get_mbx_nxt* and *get_mbx_msgs* extract a mailbox's next pointer and list of messages, respectively, from a mailbox object, using HOL primitives. For example, *get_mbx_next* looks like this in HOL:

```
let get_mbx_next = new_definition (`get_mbx_next`,
  "(∀(mbx : mbox_def).
    get_mbx_next mbx = (FST mbx)
  )"
);;
```

The HOL destructor *FST* extracts the first part of a cartesian product type. The corresponding destructor *SND* is used to extract the second part.

A mailbox can be constructed out of a separate next pointer and list of messages using the *cons_mbx* constructor:

```
let cons_mbx = new_definition (`cons_mbx`,
  "(∀ (nxt : *) (msgs : **).
    cons_mbx nxt msgs = (nxt, msgs)
  )"
);;
```

Note that this constructor uses HOL generic types * and **, and so can be used to construct any HOL cartesian type.

Constructors and destructors can be combined, as shown in this definition of *put_mbx_msg*, which appends a message to the end of a mailbox but leaves the next pointer unchanged:

```
let put_mbx_msg = new_definition (`put_mbx_msg`,
  "(∀ (mbx : mbox_def) (msg : msg_def).
     put_mbx_msg mbx msg =
       let msgs = (get_mbx_msgs mbx)
       and nxt = (get_mbx_next mbx) in
       let new_msgs = (SNOC msg msgs) in
       (cons_mbx nxt new_msgs)
  )"
);;
```

This function uses *get_mbx_nxt* and *get_mbx_msgs* to extract a mailbox's next pointer and list of messages, respectively, appends the new message to the end of the message list using *SNOC* (a HOL definition for appending an element to the end of a list), and then puts the next pointer and modified list back together using *cons_mbx*.

We also created a generic HOL theory for manipulating objects of type $((*\#**):set)$, of which *mboxes* is an instance. This theory implements some simple functions to extract the domain and range of the partial function, a consistency checker to determine if a set of (*,**) pairs is a function, a function to map a domain value to its corresponding range value, etc.

### 7.2.3 Trace Specifications for the Agents

For the two agents, traces (of the type *trace_def*) are defined as functions of type ($num \rightarrow trace\_element$). A *trace_element* is a HOL cartesian product type triplet:

$$(Agent\#TLabel\#trace\_state)$$

*Agent* denotes the agent that caused the transition and is an enumerated type consisting of $ENV$, denoting any of the agents other than agents $F$ and $FG$, $SYS\_F$, denoting agent $F$, and $SYS\_FG$, denoting agent $FG$.

The *TLabel* component (short for "transition label") is used as a shorthand method of indicating what kind of transition was made by either $F$ or $FG$. *TLabel* is an enumerated type with these possible values: ***FREADS***, ***FRESPONDS***, ***FGREADS***, ***FGRESPONDS***, and *STUTTER*. We found that it was much easier in the composition step to use the *TLabel* markers along with *Agent* than to use *Agent* and the state changes alone. The specifications for both $F$ and $FG$ insure that only the appropriate transition label is used for each transition.

The *trace_state* component is the entire system state, including the globally visible mailboxes and the local state of each agent. The trace state is defined as the cartesian product type (*mboxes#trace_istate*). The component *trace_istate* contains the local state for the two agents, and is defined as (*F_internal_state#FG_internal_state*).

The local state types vary according to the type of agent — terminal or non-terminal. The terminal agent, $F$, has the following local state:

$$(F\_work\_buf\#F\_work\_buf\_flag)$$

The buffer, *F_work_buf*, is used to store a single message after the message is read by a *read* transition from the head of the mailbox, and is of type *msg_def*. The boolean flag, *F_work_buf_flag*, is set whenever there is a message in the buffer and reset when a message is removed from the buffer.

In addition to a buffer and a buffer flag, the non-terminal agent $FG$ has a message queue that it uses to buffer messages while it is waiting for a response from $F$. The queue is defined as a list of messages. The complete local state for agent $FG$ is (*FG_buf#FG_flag#FG_queue*).

### 7.2.4 Trace Specifications for the Entry Points

For the composed system, traces (of the type $OS\_trace\_def$) are defined as functions of type ($num \rightarrow OS\_trace\_element$). An $OS\_trace\_element$ is a HOL cartesian product type triplet: ($Agent\#TLabel\#OS\_trace\_state$). The $Agent$ and $TLabel$ components are the same as for the agents, described above.

The $OS\_trace\_state$ component is defined as the HOL cartesian product type ($mboxes\#OS\_internal\_state$). The component $OS\_internal\_state$ contains the internal state for the two entry points, and is defined as

$$(OSF\_internal\_state\#OSFG\_internal\_state)$$

The two entry points in the composed system each have local state in the form of a buffer queue, which is specified as a list of messages. The HOL type of the $OSF\_queue$ and $OSFG\_queue$ components is $(msg\_def)list$.

## 7.3 Entry Point Specifications

As described in section 3.4, every specification has the form $E \Longrightarrow M$, where $E$ is the specification for the environment assumptions and $M$ is the specification for the system. In this section, we describe the specifications for $M$ in our system — the compound system that contains the two entry points.

The specification for a system is written as the property $I \cap T(N) \cap L$, where $I$ and $T(N)$ are the initial state and state transition safety properties, respectively, and $L$ is a progress property (see section 3.3). In this section, we explain each of these properties for the composed system.

The specifications of the composed system entry points both have the same general form as that described for terminal calling agents in section 5.4.1. Each performs two atomic actions: transferring a single request message from its mailbox to an internal queue (the *read transition*) and sending a response message for a single queued request message (the *respond transition*), as shown in figure 7.2, above. A progress property for each atomic action ensures that all request messages are eventually read and buffered, and that the buffered messages eventually generate responses.

## 7.3.1 Entry Point Initial State Specification

The initial state conditions for the composed system's local state is limited to specifying that the entry point queues are both empty:

```
let OS_init = new definition (`OS_init`,
  "(∀ (trace : OS_trace_def) .
    OS_init trace =
      let init_t = (trace 0) in
      let OSis = (get_OStrace_is (get_OStrace_state init_t)) in (
        ¬(OSis_Fq_pending OSis) ∧
        ¬(OSis_FGq_pending OSis))
  )"
);;
```

*OSis_Fq_pending* (resp., *OSis_FGq_pending*) returns TRUE if $F$'s ($FG$'s) local queue has a length greater than zero. This predicate is TRUE if neither of the local queues contain any messages in the initial state of a trace (*trace* 0).

Note that this initial state specification defines the initial conditions on the local state only. Initial conditions on the global system state are defined by the environment specification (described below in section 7.5.1.1).

### 7.3.2 Entry Point Read Transition Specification

The specifications for the *OSFG* and *OSF* entry point state transitions are the same except for the functions that they calculate and necessary renaming of HOL constructors and destructors. For the sake of brevity, we will show just the *OSF* state transition specifications here.

This is the specification for the *OSF* entry point read transition:

```
let OS_reads_F = new_definition (`OS_reads_F`,
  "(∀ (ss1, ss2 : OS_trace_state) .
    OS_reads_F ss1 ss2 =
     (ss2 = (OS_reads_F_msg ss1))
  )"
);;
```

The state transition is defined as a function between two successive states in a trace. The function is specified as follows:

```
let OS_reads_F_msg = new_definition (`OS_reads_F_msg`,
  "(∀ (ss : OS_trace_state) .
    OS_reads_F_msg ss =
     let mbxs = (get_OStrace_mbxs ss)
     and OSis = (get_OStrace_is ss) in
     let Fmbx = (get_mbxs_mbx mbxs F_ID) in
   %Check if no messages. If none, return state unchanged. %
      ((¬ mbx_is_unread_msg Fmbx) ⇒ ss |
   %Otherwise, read message... %
      (let (msg, new_Fmbx) = (read_mbx_msg Fmbx) in
   %put back the modified mailbox (with incremented next pointer) %
      let new_mbxs = (put_mbxs_mbx mbxs F_ID new_Fmbx)
   %put msg in the local queue %
      and new_OSis = (OSis_put_Fq_msg OSis msg) in
        (put_OStrace_mbxs (put_OStrace_is ss new_OSis)
        new_mbxs)))
  )"
);;
```

The function returns the state unchanged if the mailbox has no unread messages. If there is a message, the state that is returned is the same as the input state, except that the message at the head of *OSF*'s mailbox is put at the tail of *OSF*'s local queue.

The destructors *get_OStrace_mbxs* and *get_OStrace_is* extract the mailboxes and local state, respectively, from the trace state. The destructor *get_mbxs_mbx* extracts the mailbox associated with a specified identifier from a set of mailboxes, in this case the mailbox associated with the identifier *F_ID*. The function *mbx_is_unread_msg* checks if there are any unread messages in the mailbox by comparing the mailbox's next pointer with the length of its message list. The message list in a mailbox is indexed beginning at 0. If the numerical value of the next pointer equals the length of the list, then the list is empty.

The function *read_mbx_msg* reads the message from the head of the mailbox list and increments the next pointer, returning the message and the modified mailbox. The constructor *put_mbxs_mbx* puts the modified mailbox back into the set of mailboxes, replacing the original, while *OSis_put_Fq_msg* appends the message to the *OSF* entry point's local queue. The constructors *put_OStrace_is* and *put_OStrace_mbxs* put the modified local state and modified mailboxes together to form the system state that results from the transition.

#### 7.3.2.1 Standard Progress Property

The progress property for the *OS_reads_F* state transition is

```
let OS_reads_F_progress = new_definition ('OS_reads_F_progress',
  "(∀(trace : OS_trace_def) .
   OS_reads_F_progress =
   (∀(i : num) . ∃(j : num) .
     (i <= j) ∧
     (get_OStrace_state(trace(j+1)) =
       (OS_reads_F (get_OStrace_state(trace j)))))
  )"
);;
```

The progress property specifies that, starting at any time $i$ in the trace, there is another time $j$, either now or in the future, when the transition happens. The effect of this property is to specify that the transition happens infinitely often.

The format of this progress property is used for all progress properties in the template. From now on we will refer to it as the "standard" progress property.

### 7.3.3 Entry Point Respond Transition Specification

This is the specification for the *OSF* entry point respond transition:

```
let OS_responds_F = new definition (`OS_responds_F`,
  "(∀ (ss1, ss2 : OS_trace_state) (F_func : num → num) .
    OS_responds_F ss1 ss2  F_func =
     (ss2 = (OS_responds_F_msg ss1 F_func))
  )"
);;
```

As with the *OS_reads_F* transition, this state transition is defined as a function between two successive states in a trace. The function is specified as follows:

```
let OS_responds_F_msg = new definition (`OS_responds_F_msg`,
  "(∀ (ss : OS_trace_state) (F_func : num → num).
    OS_responds_F_msg ss F_func =
     let mbxs = (get_OStrace_mbxs ss)
     and OSis = (get_OStrace_is ss) in
  % Check if F queue has any messages...  %
  %  ...If not, return state unchanged.   %
        ((¬OSis_Fq_pending OSis) ⇒ ss |
  %Otherwise, create response message... %
        (let (rqst, new_OSis) = (OSis_Fq_get_msg OSis) in
         let src = (get_msg_sndr rqst)
         and mdata = (get_msg_data rqst)
         and mid = (get_msg_id rqst) in
  %Construct a response message, applying a function to input data %
         let response = (cons_msg F_ID (F_func mdata) mid) in
  %Put response message in destination mailbox %
         let new_mbxs = (put_mbxs_msg mbxs src response) in
         (put_OStrace_mbxs (put_OStrace_is ss new_OSis) new_mbxs)))
  )"
);;
```

The function returns the state unchanged if the local queue is empty. If there is a request message in the queue, the message at the head of the *OSF* queue is removed and a response message is put at the end of the mailbox of the sender of the original request message. The rest of the state is unchanged.

Note that the function *F_func*, applied by the transition to the input value when creating the response message, is universally quantified, and can be instantiated with any function of the correct type. This type of parameterization is what makes the specification abstract. The specification for the *OSFG* entry point similarly has the function *FG_func* as a universally quantified parameter.

The progress property for this state transition is similar to the standard progress property described for the *OS_reads_F* transition (see section 7.3.2.1) and specifies that the *OS_responds_F* transition happens infinitely often.

### 7.3.4 Entry Point State Transition Relation Specification

The complete entry point state transition relation specification for the composed, compound system (i.e., for both the *OSF* and *OSFG* entry points) has the following specification:

```
let OS_safety = new definition (‘OS_safety‘,
  "(∀ (trace : OS_trace_def) (F_func, FG_func : num → num).
    OS_safety trace F_func FG_func =
    ∀(i : num) .
     let t1 = (trace i)
     and t2 = (trace(SUC i)) in (
     ¬(get_OStrace_agent t2 = ENV) ⟹
       let ss1 = (get_OStrace_state t1)
       and ss2 = (get_OStrace_state t2) in (
  % stuttering step %
        (ss1 = ss2) ∨
  % or a legal transition %
        (((get_OStrace_tlabel t2) = FREADS) ∧ (OS_reads_F ss1 ss2)) ∨
        (((get_OStrace_tlabel t2) = FGREADS) ∧ (OS_reads_FG ss1 ss2)) ∨
        (((get_OStrace_tlabel t2) = FRESPONDS) ∧ (OS_responds_F ss1 ss2 F_func)) ∨
        (((get_OStrace_tlabel t2) = FGRESPONDS) ∧ (OS_responds_FG ss1 ss2 FG_func))))
  )"
);;
```

Recall that the universe of agents is divided into environment agents and system agents. The agents in this system have the identifiers *F_ID* and *FG_ID*. This specification constrains only the transitions that are caused by system agents. The specification says that a state transition between the state at time $i$ and the state at time $SUC(i)$ (the HOL way of saying $i + 1$) that is not caused by an environment agent must be either a stuttering step or be one of the four legal transitions, *OS_reads_F*, *OS_reads_FG*, *OS_responds_F*, or *OS_responds_FG*. Note how this entire state transition relation is parameterized by the two universally quantified functions *F_func* and *FG_func*.

## 7.4 Calling Agent Specifications

When multiple components are composed into a single system, each of the component specifications has the form $E_n \Longrightarrow M_n$, for all components $n = 0, 1, \ldots$ In this section, we describe the specifications $M_F$ and $M_{FG}$.

Like the specifications for the entry points, the specifications for the two calling agents $F$ and $FG$ have the same basic "read a message" and "send a response" transitions. But while $F$ is a terminal agent, and therefore has a specification that is much like that of the two entry points, $FG$'s "send a response" transition is considerably different.

### 7.4.1 Calling Agent Initial State Specifications

As described in section 7.2.3, the local state of the terminal agent $F$ includes a single-length message buffer and a boolean flag that is used to indicate if the buffer is empty or not. The specification for the agent's initial state is as follows:

```
let F_init = new definition (`F_init`,
  "(∀ (trace : trace_def) .
    F_init trace =
      let init_t = (trace 0) in
      let Fs = (get_trace_Fs (get_trace_state init_t)) in (
        ¬(Fs_get_flag Fs))
  )"
);;
```

which says that the boolean flag is FALSE in the initial state.

Agent *FG* has the same kind of buffer and flag as agent *F*, plus a queue used to buffer messages while *FG* is waiting for responses from *F*. This is the initial state specification for the *FG* agent:

```
let FG_init = new definition (`FG_init`,
  "(∀ (trace : trace_def) .
    FG_init trace =
      let init_t = (trace 0) in
      let FGs = (get_trace_FGs (get_trace_state init_t)) in (
        ¬(FGs_get_flag FGs) ∧
        ¬(FGs_is_pending FGs))
  )"
);;
```

which evaluates to TRUE if *FG*'s buffer flag is FALSE and if there are no messages in the queue.

### 7.4.2 Calling Agent Read Transition Specifications

The specifications for the *FG* and *F* calling agent "read message" state transitions are the same (except for necessary renaming of HOL constructors and destructors) and similar to those for the entry points (described in section 7.3.2). Here we will show just the *F* calling agent "read message" state transition specification.

```
let F_reads = new definition ('F_reads',
  "(∀ (ss1, ss2 : trace_state) .
    F_reads ss1 ss2 =
     (ss2 = (F_reads_msg ss1))
  )"
);;
```

The state transition is defined as a function between two successive states in a trace. The function is specified as follows:

```
let F_reads_msg = new definition ('F_reads_msg',
  "(∀ (ss : trace_state) .
    F_reads_msg ss =
     let (mbxs,Fs) = ((get_trace_mbxs ss),(get_trace_Fs ss)) in
     let mbx = (get_mbxs_mbx mbxs F_ID) in
  % Check if no messages or if a message already in work buffer... %
  % If so, return state unchanged. %
       (((¬mbx_is_unread_msg Fmbx) ∨ (Fs_get_flag Fs)) ⇒ ss |
  %Otherwise, read message... %
       (let (msg, new_mbx) = (read_mbx_msg mbx) in
  %put back the modified mailbox (with incremented next pointer) %
       let new_mbxs = (put_mbxs_mbx mbxs F_ID new_mbx)
  %put msg in work buffer %
      and new_Fs = (Fs_put_buf Fs msg) in
        (put_trace_mbxs (put_trace_Fs ss new_Fs) new_mbxs)))
  )"
);;
```

The function returns the state unchanged if the mailbox has no unread messages or if the single-length buffer is already full. If there is an unread message and the buffer is empty, the message at the head of $F$'s mailbox is removed from the mailbox (by *read_mbx_msg*, which does so by advancing the next pointer) and put in the buffer (done by *Fs_put_buf*, which also sets the flag). A standard progress property (described in section 7.3.2.1), *F_reads_progress*, specifies that the transition occurs infinitely often.

### 7.4.3 Calling Agent Respond Transition Specifications

The state transition where agent $F$ sends a response message is much like the response transitions of the entry points. The state transition where $FG$ sends a response, however, is quite different because *FG* must differentiate between request messages from outside the system and response messages from *F*.

### 7.4.3.1 Agent *F* Respond Transition

This is the specification for the *F* agent respond transition. It is parameterized by the function *F_func* used to calculate the response message.

```
let F_responds = new definition (`F_responds`,
  "(∀ (ss1, ss2 : trace_state) (F_func : num → num) .
    F_responds ss1 ss2 F_func =
     (ss2 = (F_responds_msg ss1 F_func))
  )"
 );;
```

This state transition is defined as a function between two successive states in a trace. The function is specified as follows:

```
let F_responds_msg = new definition (`F_responds_msg`,
  "(∀ (ss : trace_state) (F_func : num → num).
    F_responds_msg ss F_func =
     let (mbxs,Fs) = ((get_trace_mbxs ss),(get_trace_Fs ss)) in
  % Check if work buffer has a message...   %
  %  ...If not, return state unchanged.   %
      ((¬Fs_get_flag Fs) ⇒ ss |
  %Otherwise, create response message... %
      (let (rqst, new_Fs) = (Fs_get_buf Fs) in
       let src = (get_msg_sndr rqst)
       and mdata = (get_msg_data rqst)
       and mid = (get_msg_id rqst) in
  %Construct a response message, applying a function to input data %
       let response = (cons_msg F_ID (F_func mdata) mid) in
  %Put response message in destination mailbox %
       let new_mbxs = (put_mbxs_msg mbxs src response) in
         (put_trace_mbxs (put_trace_Fs ss new_Fs) new_mbxs)))
  )"
 );;
```

The function returns the state unchanged if the local buffer is empty (because there is nothing to respond to). If there is a request message in the buffer, the message is removed from the buffer (the value *new_Fs* returned by *Fs_get_buf* has the flag reset) and a response message is put at the end of the mailbox of the sender of the original request message. The rest of the state is unchanged. A standard progress property, *F_responds_progress*, asserts that this transition happens infinitely often.

#### 7.4.3.2 Agent *FG* Respond Transition

As described in section 5.4.2, because a non-terminal calling agent has only one mailbox, it must continue to read messages while it is waiting for a response from the agent or agents that it calls. For this reason the "send a response" state transition for the non-terminal agent is considerably different from that of the terminal agent. A review of section 5.4.2 may be helpful in understanding this section. The data path between the *F* and *FG* calling agents is shown in figure 7.4, where the solid arrows show the path up until the response message from agent *F* and the dotted lines show the path after that point.

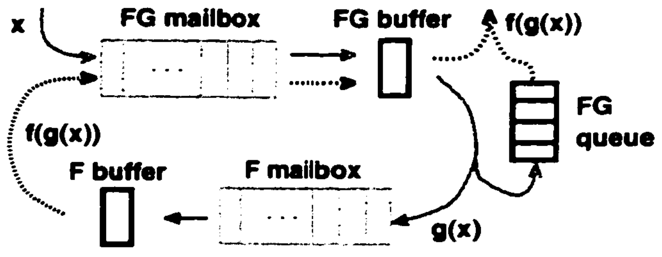

Figure 7.4: Data path for FG responses

The figure shows a request message that contains the data value $x$ arriving in *FG*'s mailbox. The solid line between the *FG* mailbox and the *FG* buffer represents a *read* transition by the *FG* agent. The solid line that starts from the *FG* buffer and splits between the *FG* queue and the *F* mailbox represents the *FG* agent queueing the request and sending a request message to the *F* calling agent. This request message contains the data value $g(x)$. The small solid line between the $F$ mailbox and the $F$ buffer represents a *read* transition by the $F$ calling agent. The dotted line between the $F$ buffer and the $FG$ mailbox represents the $F$ agent sending a response message to the $FG$ agent. The dotted line between the $FG$ mailbox and the $FG$ buffer represents another *read* transition by the $FG$ agent, and the split dotted line that originates at both the $FG$ buffer and the $FG$ queue represents the $FG$ agent sending a response message.

This is the specification for the $FG$ agent respond transition. It is parameterized by the function *G_func* used to calculate the response message.

```
let FG_responds = new definition (`FG_responds`,
  "(∀ (ss1, ss2 : trace_state) (G_func : num → num) .
    FG_responds ss1 ss2 G_func =
     (ss2 = (FG_responds_msg ss1 G_func))
  )"
 );;
```

This state transition is defined as a function between two successive states in a trace. The function is specified as follows:

```
let FG_responds_msg = new definition (`FG_responds_msg`,
  "(∀ (ss : trace_state) (G_func : num → num).
    FG_responds_msg ss G_func =
     let (mbxs, FGs) = ((get_trace_mbxs ss), (get_trace_FGs ss)) in
  % Check if work buffer has a message...  %
  %  ...If not, return state unchanged.  %
       ((¬FGs_get_flag FGs) ⇒ ss |
  %Otherwise, check who sent the message... %
       (let (bufmsg, FGs1) = (FGs_get_buf FGs) in
        let bufsrc = (get_msg_sndr bufmsg) in
  %If sender is not F, queue message and send request to F %
       (¬(bufsrc = F_ID) ⇒ (FG_push_request ss G_func) |
  %Otherwise, F sent the message so send a response message %
        (FG_send_response ss))))
  )"
 );;
```

As with all of the other state transitions described above, a standard progress property, *FG_responds_progress* asserts that this transition happens infinitely often.

The function returns the state unchanged if the local buffer is empty. If there is a request message in the buffer, *FG* takes different actions that depend on the identity of the message sender. If the sender is not the terminal agent, the message is a request message from outside the system. In this case, the agent queues the request message and sends a request to agent *F*. These actions are performed by *FG_push_request*, described below.

If the message was sent by *F*, the message is a response from *F* to a previous request sent by *FG*. The value contained in the message from *F* is the value that is to be returned to the sender of the original request message at the head of *FG*'s local queue. In this case, *FG* removes the message from the head of the queue, determines who the original sender was, and then sends a response message using the value sent by *F*. These actions are performed in *FG_send_response*.

**FG_push_request** Here is the specification for *FG_push_request*:

```
let FG_push_request = new definition ('FG_push_request',
  "(∀ (ss : trace_state) (G_func : num → num).
    FG_push_request ss G_func =
     let (mbxs, FGs) = ((get_trace_mbxs ss),(get_trace_FGs ss)) in
     let (bufmsg, FGs1) = (FGs_get_buf FGs) in %FGs1 has cleared buffer %
   % Put request message in the queue %
     let new_FGs = (FGs_put_queue_msg FGs1 bufmsg) in
   % Create a message to send to F %
     let bufmdata = (get_msg_data bufmsg)
     and bufmid = (get_msg_id bufmsg) in
     let msg_for_F = (cons_msg FG_ID (G_func bufmdata) bufmid) in
   % Send the message to F %
     let new_mbxs = (put_mbxs_msg mbxs F_ID msg_for_F) in
       (put_trace_mbxs (put_trace_FGs ss new_FGs) new_mbxs)
  )"
 );;
```

This function puts the request message at the end of the queue (*FGs_put_queue_msg*), extracts the data from the request message (*bufmdata*), creates a message to send to *F* containing the value *G_func*(*bufmdata*), and then sends the request message to *F*. *F* will eventually read the request message from $FG$ and respond to $FG$ with a message containing the data value $F\_func(G\_func(bufmdata))$.

When a message in $FG$'s buffer is from $F$, the message was sent by $F$ in response to a request message previously sent by $FG$. $F$ is specified to respond to messages in the order that they are put into $F$'s mailbox, so response messages from $F$ to $FG$ will be received by $FG$ in the same order that $FG$ sent the request messages to $F$ (we prove this as an invariant).

**FG_send_response** This is the specification for *FG_send_response*:

```
let FG_send_response = new definition (`FG_send_response`,
  "(∀ (ss : trace_state).
   FG_send_response ss =
    let (mbxs, FGs) = ((get_trace_mbxs ss),(get_trace_FGs ss)) in
    let (bufmsg, FGs1) = (FGs_get_buf FGs) in %FGs1 has cleared buffer %
     ((FGs_is_pending FGs1) ⇒
       (let (rqst, new_FGs) = (FGs_get_queue_msg FGs1) in
        let bufmdata = (get_msg_data bufmsg) in
        let rsrc = (get_msg_sndr rest)
        and rmid = (get_msg_id rqst) in
  % Create response message %
        let response = (cons_msg FG_ID bufmdata rmid) in
          (put_trace_mbxs (put_trace_FGs ss new_FGs) new_mbxs)) |
       (put_trace_FGs ss FGs1))
  )"
);;
```

This specification first checks that there is something in the queue (*FGs_is_pending*). If not, it simply returns the state with the buffer empty. This is done for completeness, but the system specification never permits this situation to occur (we prove this as an invariant). If there is a message in the queue, the sender's identifier (and message id, although the message id is not otherwise used) are extracted from the message at the head of the queue. A response message that includes the original sender's message id and the data sent by $F$ (whose message is in the buffer) is created. The agent sends the response message to the original sender of the queued message. The queued message is removed from the queue and the buffer is emptied.

### 7.4.4 Calling Agent State Transition Relation Specification

The complete calling agent state transition relation specification for agent $F$ is as follows:

```
let F_safety = new definition ('F_safety',
  "(∀ (trace : trace_def) (F_func : num → num).
   F_safety trace F_func =
   ∀(i : num) .
   let t1 = (trace i)
   and t2 = (trace(SUC i)) in
    (get_trace_agent t2 = SYS_F) ⟹
     let ss1 = (get_trace_state t1)
     and ss2 = (get_trace_state t2) in
  % stuttering step %
       (((get_trace_tlable t2) = STUTTER) ∧ (ss1 = ss2)) ∨
  % or a legal transition %
       ((((get_trace_tlabel t2) = FREADS) ∧ (F_reads ss1 ss2)) ∨
       ((((get_trace_tlabel t2) = FRESPONDS) ∧ (F_responds ss1 ss2 F_func))))
  )"
);;
```

This specification says that all transitions caused by the agent $SYS\_F$ must be either a stuttering step or one of the two legal transitions defined for that agent. The state transition relation for agent $FG$ is the same, except for the obvious renaming, and is omitted here.

## 7.5 Environment Specifications

Our template composes two specifications $E_F \Longrightarrow M_F$ and $E_{FG} \Longrightarrow M_{FG}$ into the system $E_{OS} \Longrightarrow M_{OS}$. In previous sections, we have covered $M_{OS}$, $M_F$, and $M_{FG}$. In this section, we will describe $E_{OS}$, $E_F$, and $E_{FG}$. Note that there are no progress properties on the environment — the environment *may* send request messages, but it does not have to.

### 7.5.1 Composed System Environment Assumptions

In this section, we describe the initial global state specification and the environment state transitions.

#### 7.5.1.1 System Environment Initial State Specification

The initial state specification for the composed system environment is the initial global state specification (the environment has no local state) for the entire composed system specification, including the entry points. The initial conditions on the global state consist of the initial values of the state variables and on their structural attributes.

First of all, each of the agents $F$ and $FG$ must have a mailbox in the set of mailboxes. Mailboxes are specified to exist for only agents $F$ and $FG$ because there is no need to put a limit on the other agents that can exist in the environment. A benefit of leaving the other agents unspecified is that the composed system can more easily be composed with other components.

The identifiers for the two agents must be distinct. The set of mailboxes, furthermore, must implement a partial function from identifiers to mailboxes (i.e., there can be no duplicate identifiers) so that every agent has only one mailbox and the next pointer for each of the mailboxes must be less than or equal to the length of that mailbox.

The initial state global state values are that every mailbox is empty. This is specified by requiring that there are no unread messages in any of the mailboxes.

## 7.5.2 System Environment State Transitions

The composed system environment state transitions are (assumed to be) limited to four types of atomic actions:

1. a stuttering step,
2. an agent in the environment may send a message to $F$'s mailbox,
3. an agent in the environment may send a message to $FG$'s mailbox,
4. anything else, so long as the environment leaves $F$'s and $FG$'s mailboxes alone.

These options are specified as follows:

```
let OS_env_safety = new definition ('OS_env_safety',
  "(∀ (trace : trace_def).
    OS_env_safety trace =
     ∀(i : num) .
     let t1 = (trace i)
     and t2 = (trace(SUC i)) in (
      (¬(get_trace_agent t2 = SYS_F) ∧
       ¬(get_trace_agent t2 = SYS_FG)) ⟹
       let ss1 = (get_trace_state t1)
       and ss2 = (get_trace_state t2) in (
  % stuttering step %
        (ss1 = ss2) ∨
  % or a legal transition %
        (OS_env_SEND_F ss1 ss2) ∨
        (OS_env_SEND_F ss1 ss2) ∨
        (OS_env_arb ss1 ss2)))
  )"
 );;
```

This specification says that, for any state transition not caused by the agents $SYS\_F$ and $SYS\_FG$, the transitions are assumed to be as listed. Note that there are no assertions about the trace labels, as there are for the state transitions that are part of the system. This is because, when we map the environment transitions during the refinement step, we do not care which of the environment agents caused, for example, a stuttering step. We care only that it was not either of the two agents.

The careful reader will also note that this specification used the same trace definition (*trace_def*) used by the agent specifications and not the definition (*OS_trace_def*) used in the entry point specifications. The two differ only in the local state definitions and because the environment is specified to never alter the local state of the system agents, it is simple to extract just the global state from a trace definition (of either type). We could have used *OS_trace_def*, but having the overall system environment specification use the same trace definition as the environments for the two calling agents was convenient, albeit at the loss of some clarity in this explanation.

#### 7.5.2.1 System Environment *SEND* State Transitions

Unlike the state transitions for the calling agents and entry points, which are specified as functions from state to state, the state transitions for the environment are specified as relations. If we were to limit the set of environment agents and exhaustively enumerate how each of the environment agents could send a message to a system agent, we could use functions. A relation, however, is simpler because we simply have to say that *some* message arrives from *any* environment agent.

Here is the specification for the state transition where the environment sends a message to the $F$ mailbox:

```
let OS_env_SEND_F = new definition (`OS_env_SEND_F`,
  "(∀ (ss1, ss2 : trace_state).
    OS_env_SEND_F ss1 ss2 =
    let (mbxs1, mbxs2) = ((get_trace_mbxs ss1), (get_trace_mbxs ss2))
    and internals1 = (get_trace_not_mbxs ss1)
    and internals2 = (get_trace_not_mbxs ss2) in
    let Fmbx1 = (get_mbxs_mbx mbxs1 F_ID)
    and Fmbx2 = (get_mbxs_mbx mbxs2 F_ID) in
    let FGmbx1 = (get_mbxs_mbx mbxs1 FG_ID)
    and FGmbx2 = (get_mbxs_mbx mbxs2 FG_ID) in (
  %No new mailboxes can be created nor a mailbox be deleted %
      ((dom mbxs1) = (dom mbxs2)) ∧
  %The internal state must be untouched %
      (internals1 = internals2) ∧
  %The FG mailbox must be untouched %
      (FGmbx1 = FGmbx2) ∧
  %There must be a new message in the F mailbox %
      (∃(msg : msg_def) .
        let sndr = (get_msg_sndr msg) in (
  %The new message's sender must have a mailbox %
        (sndr IN (dom mbxs1)) ∧
  %The sender's id cannot be F_ID or FG_ID (system agents) %
        ¬(sndr = F_ID) ∧
        ¬(sndr = FG_ID) ∧
  %The new message must be at the end of the F mailbox... $
  % ...all other messages and their order untouched       %
        ((put_mbx_msg Fmbx1 msg) = Fmbx2))))
  )"
);;
```

This specification says that, as a result of the transition, there must be a new message from a legitimate environment agent (not $F$ or $FG$, which are system agents), that the sender must have a mailbox, and that $F$'s mailbox must be untouched except for the new message at the end of it. The domain of mailboxes, furthermore, all local state, and the $FG$ mailbox, must be left unchanged by the transition (the environment may do what it pleases with the other mailboxes as that does not affect $F$ and $FG$). The specification for the state transition where the environment sends a message to the $FG$ mailbox is the same except for simple renaming.

#### 7.5.2.2 System Environment *Arbitrary* State Transitions

The *arbitrary* environment state transitions are so-called because they put almost no restrictions on what the environment is assumed to do. The only restrictions are that the environment neither creates nor deletes any mailboxes, that the $F$ and $FG$ mailboxes are left untouched, and that the local state does not change. This is specified as follows:

```
let OS_env_arb = new definition ('OS_env_arb',
  "(∀ (ss1, ss2 : trace_state).
    OS_env_arb ss1 ss2 =
     let (mbxs1, mbxs2) = ((get_trace_mbxs ss1), (get_trace_mbxs ss2))
     and internals1 = (get_trace_not_mbxs ss1)
     and internals2 = (get_trace_not_mbxs ss2) in
     let Fmbx1 = (get_mbxs_mbx mbxs1 F_ID)
     and Fmbx2 = (get_mbxs_mbx mbxs2 F_ID) in
     let FGmbx1 = (get_mbxs_mbx mbxs1 FG_ID)
     and FGmbx2 = (get_mbxs_mbx mbxs2 FG_ID) in (
  %No new mailboxes can be created nor a mailbox be deleted %
      ((dom mbxs1) = (dom mbxs2)) ∧
  %The internal state must be untouched %
      (internals1 = internals2) ∧
  %The F and FG mailboxes must be untouched %
      (Fmbx1 = Fmbx2) ∧
      (FGmbx1 = FGmbx2))
  )"
 );;
```

### 7.5.3 Agent Environment Assumptions

In this section, we describe the agent environment assumption specifications, $E_F$ and $E_{FG}$.

#### 7.5.3.1 Agent Environment Initial State Specifications

The environment assumption initial state specification for agent $F$ is almost identical to the overall system environment assumption initial state specification. The only difference is that $F$'s initial assumption specifies only that $F$ must have a mailbox; it does not mention *FG* at all. This is because, from *F*'s point of view, *FG* is part of *F*'s environment.

The environment assumption initial state specification for agent *FG* is the same as that for the overall system environment. Although *F* is part of *FG*'s environment, *FG*'s environment assumption must specify that a mailbox exists for *F* as well as for *FG* because *FG* sends request messages to *F*.

### 7.5.4 Agent Environment State Transitions

The state transitions for the agent environments are identical except for obvious renaming, so we will show only those for agent *F*'s environment:

```
let F_env_safety = new definition ('F_env_safety',
  "(∀ (trace : trace_def).
    F_env_safety trace =
    ∀(i : num) .
    let t1 = (trace i)
    and t2 = (trace(SUC i)) in (
      ¬(get_trace_agent t2 = SYS_F) ⟹
      let ss1 = (get_trace_state t1)
      and ss2 = (get_trace_state t2) in (
  % stuttering step %
        (ss1 = ss2) ∨
  % or a legal transition %
        (F_env_SEND ss1 ss2) ∨
        (F_env_arb ss1 ss2)))
  )"
);;
```

From *F*'s point of view, any changes to *FG*'s mailbox fall into the "arbitrary" category of environment transitions, so there is no constraint on what changes can happen to that mailbox.

#### 7.5.4.1 Agent Environment *SEND* State Transitions

Like the system environment state transitions, the state transitions for the agent environment are specified as relations. Here is the specification for the state transition where the environment sends a message to the *F* mailbox:

```
let F_env_SEND = new definition (`F_env_SEND`,
  "(∀ (ss1, ss2 : trace_state).
    F_env_SEND ss1 ss2 =
     let (mbxs1, Fs1) = ((get_trace_mbxs ss1), (get_trace_Fs ss1))
     and (mbxs2, Fs2) = ((get_trace_mbxs ss2), (get_trace_Fs ss2))in
     let Fmbx1 = (get_mbxs_mbx mbxs1 F_ID)
     and Fmbx2 = (get_mbxs_mbx mbxs2 F_ID) in (
  %No new mailboxes can be created nor a mailbox be deleted %
       ((dom mbxs1) = (dom mbxs2)) ∧
  %The internal state must be untouched %
       (Fs1 = Fs2) ∧
  %There must be a new message in the F mailbox %
       (∃(msg : msg_def) .
         let sndr = (get_msg_sndr msg) in (
  %The new message's sender must have a mailbox %
           (sndr IN (dom mbxs1)) ∧
  %The sender's id cannot be F_ID (viz., the system agent) %
           ¬(sndr = F_ID) ∧
  %The new message must be at the end of the F mailbox... $
  % ...all other messages and their order untouched       %
           ((put_mbx_msg Fmbx1 msg) = Fmbx2))))
  )"
);;
```

This specification says that, as a result of the transition, there must be a new message from a legitimate environment agent (not *F*, which is a system agent), that the sender must have a mailbox, and that *F*'s mailbox must be untouched except for the new message appended to it. The domain of mailboxes, furthermore, and *F*'s local state must be left unchanged by the transition (the environment may do what it pleases with the other mailboxes as that does not affect *F*). The specification for the state transition where the environment sends a message to the *FG* mailbox is the same except for simple renaming.

#### 7.5.4.2 Agent Environment *Arbitrary* State Transitions

The only restrictions in the *F* agent's environment arbitrary transition are that the environment neither creates nor deletes any mailboxes, that the *F* mailbox is left untouched, and that *F*'s local state does not change. This is specified as follows:

```
let F_env_arb = new definition ('F_env_arb',
  "(∀ (ss1, ss2 : trace_state).
   F_env_arb ss1 ss2 =
    let (mbxs1, Fs1) = ((get_trace_mbxs ss1), (get_trace_Fs ss1))
    and (mbxs2, Fs2) = ((get_trace_mbxs ss2), (get_trace_Fs ss2))in
    let Fmbx1 = (get_mbxs_mbx mbxs1 F_ID)
    and Fmbx2 = (get_mbxs_mbx mbxs2 F_ID) in (
  %No new mailboxes can be created nor a mailbox be deleted %
      ((dom mbxs1) = (dom mbxs2)) ∧
  %The internal state must be untouched %
      (Fs1 = Fs2) ∧
  %The F and FG mailboxes must be untouched %
      (Fmbx1 = Fmbx2))
  )"
);;
```

*FG*'s environment arbitrary transition is the same, except for renaming.

## 7.6 Composition Step

Now that we have introduced the specifications for the agents and the composed system, we are ready to examine the composition proof of the template. As described in chapter 3, the proof consists of two steps, the composition step and the refinement step. In this section, we discuss the composition step.

The composition proof rule is described in section 3.6. Here is how our specifications satisfy the proof rule conditions:

*If $\mu_1$, $\mu_2$, and $\mu_1 \cup \mu_2$ are agent sets...*

In our template, $\mu_1$ and $\mu_2$ are *SYS_F* and *SYS_FG*.

*...and $E$, $E_1$, $E_2$, $M_1$, and $M_2$ are properties such that:*

1. *$E = I \cap P$, $E_1 = I_1 \cap P_1$, and $E_2 = I_2 \cap P_2$, where*
   (a) *$I$, $I_1$, and $I_2$ are state predicates.*
   (b) *$P$, $P_1$, and $P_2$ are safety properties that constrain at most $\neg(\mu_1 \cup \mu_2)$, $\neg\mu_1$, and $\neg\mu_2$, respectively.*
2. *$\overline{M_1}$ and $\overline{M_2}$ constrain at most $\mu_1$ and $\mu_2$, respectively.*
3. *$\mu_1 \cap \mu_2 = \emptyset$*

$I \cap P$ in the template is expressed as

$$((OS\_env\_init\ trace) \wedge (OS\_env\_safety\ trace))$$

$I_1 \cap P_1$ is expressed as

$$((F\_env\_init\ trace) \wedge (F\_env\_safety\ trace))$$

and $I_2 \cap P_2$ is expressed as

$$((FG\_env\_init\ trace) \wedge (FG\_env\_safety\ trace))$$

*OS_env_init*, *F_init*, and *FG_init* are clearly state properties (viz., on the initial states of traces). Similarly, *OS_env_safety*, *F_env_safety*, and *FG_env_safety* are safety properties that constrain $\neg(SYS\_F \cup SYS\_FG)$, $\neg(SYS\_F)$, and $\neg(SYS\_FG)$, respectively.

$\overline{M_1}$ and $\overline{M_2}$ are expressed as $(F\_init \wedge F\_safety)$ and $(FG\_init \wedge FG\_safety)$. They constrain only $SYS\_F$ and $SYS\_F$, respectively. $SYS\_F$ and $SYS\_F$ are distinct values of a HOL enumerated type (that consists exclusively of $ENV$, $SYS\_F$ and $SYS\_F$).

*...then the rule of inference*

$$\frac{E \cap \overline{M_1} \cap \overline{M_2} \subseteq E_1 \cap E_2}{(E_1 \Rightarrow M_1) \cap (E_2 \Rightarrow M_2) \subseteq (E \Rightarrow M_1 \cap M_2)}$$

*is sound.*

Having met the soundness conditions for the rule of inference, we are ready to prove the antecedent of the rule of inference. In our template, the antecedent is expressed as the following theorem to be proven:

```
let composeF_FG = prove_thm(`composeF_FG`,
 (∀ (trace : trace_def) (F_func, G_func: num → num).
   ((OS_env_init trace) ∧ (OS_env_safety trace) ∧
    (F_init trace) ∧ (F_safety trace F_func) ∧
    (FG_init trace) ∧ (FG_safety trace G_func)) ⟹
   ((F_env_init trace) ∧ (F_env_safety trace) ∧
    (FG_env_init trace) ∧
      (FG_env_safety trace))
);;
```

Once this has been proven, we can apply the composition rule.

## 7.6.1 Proving the Composition Rule Antecedent

The complicated looking antecedent proof can be broken down into three smaller proofs:

1. **The initial state properties on the left side of the implication imply the initial state properties on the right side.** I.e., that

   ```
   ((OS_env_init trace) ∧ (F_init trace) ∧
    (FG_init trace)) ⟹
   ((F_env_init trace) ∧ (FG_env_init trace))
   ```

   The state transitions cannot affect the initial state properties, and so they play no part is this sub-proof.

   Because *F_init* and *FG_init* strictly constrain the local state of the two agents, this sub-proof reduces even further to

   ```
   (OS_env_init trace) ⟹
   ((F_env_init trace) ∧ (FG_env_init trace))
   ```

2. **The environment and *FG* state transitions imply the *F* environment state transitions.** I.e., that

```
   ((OS_env_init trace) ∧ (OS_env_safety trace) ∧
    (FG_safety trace G_func)) ⟹
   (F_env_safety trace)
);;
```

   The agent set in the *F* specification is disjoint from the agent set in *F*'s environment, so *F*'s state transitions play no part in this sub-proof. This sub-proof requires showing that each of the transitions in *OS_env_safety* and *FG_safety* satisfy the permitted transitions in *F_env_safety*. This is an example of the compositional proof complexity that we discussed in section 4.6.1.

3. **That the environment and *F* state transitions implement the *FG* environment state transitions.** I.e., that

```
   ((OS_env_init trace) ∧ (OS_env_safety trace) ∧
    (F_safety trace G_func)) ⟹
   (FG_env_safety trace)
);;
```

   This proof is symmetric with that for *F_env_safety*, except for the obvious renaming.

## 7.7 Refinement

A refinement mapping is a function from states in one specification to states in another. Because we are mapping traces, however, not just states, we must also show that the state transitions of the first specification map to state transitions (or stuttering steps) in the second, and that all progress properties are satisfied.

In this section, we describe our refinement mapping between the composed system specification and the calling agent specifications. A review of section 5.4.2.3 — a high-level discussion of the refinement of a similar type of system — may be helpful in understanding this section.

## 7.7.1 Mapping the States

The mapping function between the calling agent and composed system specifications has three parts:

1. Mapping the global state (viz., the mailboxes).
2. Mapping the *F* calling agent internal buffer to the *F* entry point internal queue.
3. Mapping the *FG* calling agent internal buffer and queue to the *FG* entry point internal queue.

### 7.7.1.1 Mapping the Mailboxes

In the calling agent specifications, *FG* sends messages to, and receives them from, *F*. These messages appear in the *F* and *FG* mailboxes and in the calling agents' internal buffers. In the calling agent specification, however, there are no individual calling agents, only the `F` and `FG` entry points, and there are no state transitions that could account for the appearance in the mailboxes of the messages passed between the calling agents. For this reason, our refinement mapping hides the messages that are passed between the two servers, leaving all other messages and their ordering alone. This filtering is depicted in figure 7.5, which shows how a message from the *FG* calling agent is filtered in the the *F* mailbox.

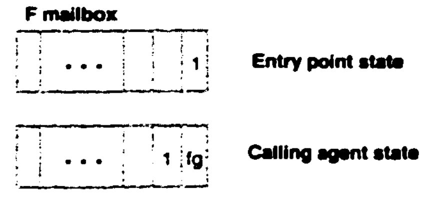

Figure 7.5: Mapping the mailboxes

#### 7.7.1.2 Mapping *F*'s State

Mapping the *F* calling agent internal buffer to the *F* entry point internal queue requires filtering similar to that of the mailboxes: any message from the *FG* calling agent is filtered, which means that the single-length buffer either maps to an empty *F* calling agent internal queue or to a queue with only one message in it. This is depicted in figure 7.6. In the figure, when the calling agent buffer contains message "2" that was sent by an agent other than *FG*, it maps to the entry point queue that contains only that same, single message. When, however, the calling agent buffer contains a message from the agent *FG*, that maps to an empty queue in the entry point.

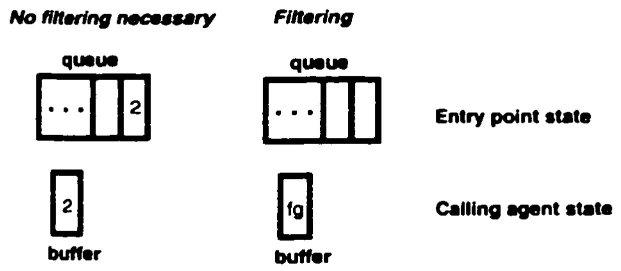

Figure 7.6: Mapping *F*'s state

#### 7.7.1.3 Mapping *FG*'s State

The *FG* calling agent saves each message in a local queue until it receives a response message from the *F* calling agent. The concatenation of the *FG* server buffer with the queue contains the same messages in the same order as in the *FG* calling agent internal queue. The mapping does this concatenation, but also filters any messages from the *F* calling agent that might be in *FG*'s buffer. This is depicted in figure 7.7, which shows one case where filtering is unnecessary and one where the contents of the buffer must be filtered.

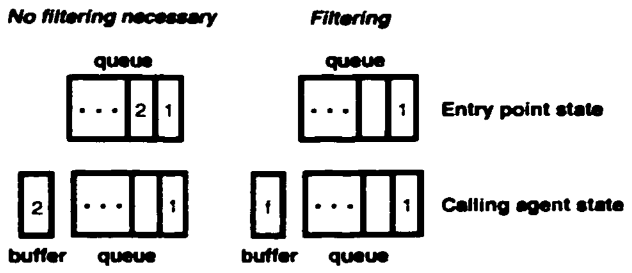

Figure 7.7: Mapping *FG*'s state

### 7.7.2 Mapping the State Transitions

Having defined a mapping function, our next step is to show that each of the calling agent state transitions map to a valid state transition, or a stuttering step, by the entry points. We need to show that the calling agent state transitions map for all possible state conditions.

#### 7.7.2.1 Mapping the *F_reads* Transition

The *F_reads* transition has three possible cases:

1. The mailbox is empty or there is already something in the buffer. The *F_reads* transition leaves the state unchanged, which maps to a stuttering step.

2. The buffer is empty and the message at the head of the mailbox is from the *FG* server. The message from the *FG* server is filtered by the mapping function, so this transition also maps to a stuttering step.

3. The buffer is empty and the message at the head of the mailbox is not from the *FG* server. This case is the only one where the *F_reads* transition maps to an *OS_reads_F* transition that does not stutter.

#### 7.7.2.2 Mapping the *F_responds* Transition

The *F_responds* transition also has three cases:

1. The buffer is empty. The *F_responds* transition leaves the state unchanged, which maps to a stuttering step.

2. The message in the buffer is from the *FG* server. The message from the *FG* server is filtered by the mapping function, so this transition also maps to a stuttering step.

3. The message in the buffer is not from the *FG* server. This case is the only one where a *F_responds* transition maps to a non-stuttering *OS_responds_F* transition.

#### 7.7.2.3 Mapping the *FG_reads* Transition

The cases of the *FG_reads* transition are the same as that of the *F_reads* transition, described above. The proof is the same except for renaming.

#### 7.7.2.4 Mapping the *FG_responds* Transition

The *FG_responds* transition also has three cases:

1. The buffer is empty. The transition leaves the state unchanged, which maps to a stuttering step.

2. The message in the buffer is not from the *F* server. Only messages from the *F* server result in a response message being sent. All other messages are request messages and are transferred to the internal queue. The request message that *FG* sends to *F* as a result of reading a request message is filtered by the mapping function. The mapping function concatenates *FG*'s buffer and queue, so no change to the mapped state occurs, and the calling agent transition maps to a stuttering step.

3. The message in the buffer is from the *F* server. This case is the only one where a *FG_responds* transition maps to a non-stuttering *OS_responds_FG* transition.

The mapping of the *FG_responds* transition is shown by proving the following theorem:

```
let map_up_FG_responds = prove_theorem(`map_up_FG_responds`,
"(∀ (trace : trace_def) (F_func G_func FG_func : num→num).
  (∀ (x : num). assumption F_func G_func FG_func x) ∧
  OS_env_init trace ∧
  OS_env_safety trace ∧
  F_init trace ∧ FG_init trace ∧
  F_safety trace F_func ∧
  FG_safety trace G_func ∧
  (FG_responds(get_trace_state(trace i))(get_trace_state(trace(SUC i))) G_func)
  ==>
  % map to a stuttering step %
  ((get_OStrace_state(map_up_element(trace i)) =
    get_OStrace_state(map_up_element(trace(SUC i)))) ∨
  % or map to an OS_responds_FG transition %
   (OS_responds_FG
     (get_OStrace_state(map_up_element(trace i)))
     (get_OStrace_state(map_up_element(trace(SUC i))))
     FG_func))
 )"
);;
```

In order to prove this theorem, the antecedent *assumption*, a predicate on the functions *F_func*, *G_func*, and *FG_func*, must first be proven. We will explain the use of *assumption* in the next section.

## 7.8 Functional Composition

The *FG_responds* transition applies the function *G_func* to the data in the request messages that it receives. The *OS_responds_FG* transition applies the function *FG_func* to the data in the request messages that it receives. These functions are not equivalent, so how can we prove that *FG_responds* sometimes maps up to *OS_responds_FG*?

To answer that, we must take into account that the *F* agent is applying the function *F_func* to the data in the request messages that it gets. The value in the response messages sent by *FG_responds* is not $G\_func(x)$, therefore, but $F\_func(G\_func(x))$. The mapping, therefore, can only be shown if we also show that $(F\_func(G\_func(x)) = FG\_func(x))$.

In the template, the specific functions $F\_func$ and $G\_func$ used by the $F$ and $G$ servers, and the function *FG_func* used by the *OSFG* entry point, are parameterized, so that any functions of type ($num \to num$) can be used provided that $F\_func \circ G\_func = FG\_func$. This requirement is precisely what *assumption* is there to ensure:

```
let assumption = new_definition(`assumption`,
 "(∀ (F_func G_func FG_func : num→num) (x : num).
  assumption F_func G_func FG_func x =
    (F_func(G_func x) = FG_func x)
 )"
);;
```

Just as the three functions are parameters to the template, *assumption* is a proof obligation that must also be supplied to the template as a parameter.

## 7.9 Mapping the Progress Properties

The proof that the mapping satisfies the progress properties requires a much more complex proof than that of the state transitions. In this section, we described the special characteristics of progress properties that make them so difficult to map.

### 7.9.1 Conditions for Mapping Progress Properties

Our specifications have simple progress properties that guarantee the eventual occurrence of each transition. The eventual occurrence of a calling agent transition, however, does not always guarantee the eventual occurrence of an entry point transition under the refinement mapping.

For example, if there is at least one message in the $F$ mailbox that is not from the $FG$ calling agent, but there is also a message in $F$'s buffer, the $F\_reads$ transition will implement a stuttering step because the precondition for the transition requires the buffer to be empty. The corresponding transition in the entry point specification, *OS_reads_F*, however, is enabled because there is a message in the mailbox. This is shown in figure 7.8.

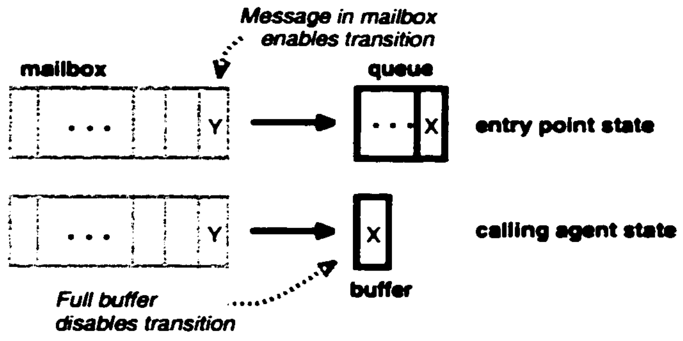
Message in mailbox enables transition
mailbox
queue
Y
X
entry point state
Y
X
calling agent state
Full buffer disables transition
buffer

Figure 7.8: Non-mapping of F_reads transition, case 1

In a second case, where *F*'s buffer is empty but there is at least one message in the *F* mailbox that is not from *FG* and the message at the head of the mailbox is from *FG*, then the preconditions are satisfied for both *F_reads* and *OS_reads_F*, but the calling agent transition, nevertheless, does not map to the entry point transition. This is because the refinement mapping filters out the messages from *FG* so the message at the head of the mailbox in the calling agent state is not the same message as at the head of the mailbox in the entry point state. This situation is depicted in figure 7.9.

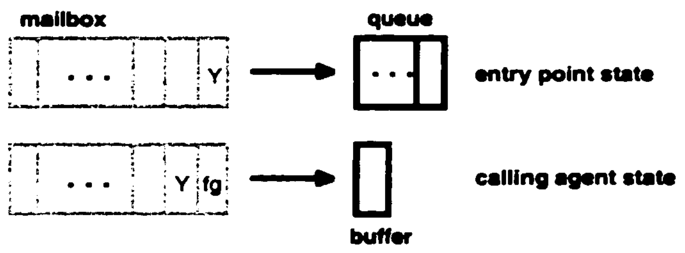

Figure 7.9: Non-mapping of F_reads transition, case 2

If, on the other hand, the only messages in the mailbox are from *FG*, then the *OS_reads_F* transition is disabled (viz., because its mailbox is empty) but the *F_reads* transition may or may not be enabled (depending on whether or not its buffer is empty), as shown in figure 7.10. If the buffer is full, the *F_reads* transition does a stuttering step. If the buffer is empty it moves the message from *FG* at the head of the mailbox into the buffer. In either case, the *F_reads* transition maps to an *OS_reads_F* transition, but the *OS_reads_F* transition does nothing (stutters).

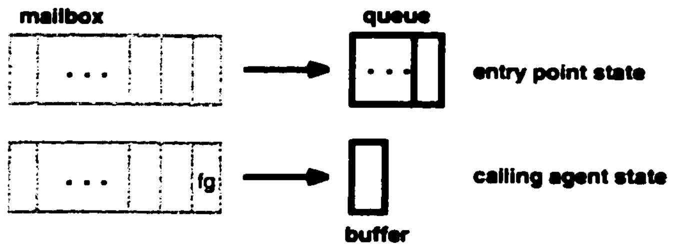

Figure 7.10: Non-mapping of F_reads transition, case 3

In a fourth case, the *F* mailbox is completely empty, so a *F_reads* transition does a stuttering step, as does the *OS_reads_F* transition to which it maps. As with the preceding case, this is true whether or not the buffer is full.

The only conditions where an enabled *F_reads* transition maps to an enabled *OS_reads_F* transition is when there is at least one message in the mailbox, *F*'s buffer is empty, and the message at the head of the mailbox is not from *FG*.

If we call these last three cases when a *F_reads* transition maps to an *OS_reads_F* transition $\rho$ and denote the *F_reads* transition and *OS_reads_F* transition as $t_L$ and $t_H$, respectively, then to prove $\Box\Diamond t_H$ (always eventually $t_H$) — the progress property on $t_H$ — we need to prove $\Box\Diamond(\rho \wedge t_L)$.

The progress properties on each transition guarantee that the transitions will occur, but not that the preconditions will ever be true. The preconditions could be true infinitely often, but never true when the transition occurs. Only if the preconditions become true and remain true until the transition occurs can we show that a transition will ever implement anything other than a stuttering step. In order to obtain $\Box\Diamond(\rho \wedge t_L)$, therefore, we need the following three conditions:

1. That $t_L$ occurs infinitely often, i.e., $\Box\Diamond t_L$. This is precisely the progress property on $t_L$.

2. That $\rho$ always eventually holds, i.e., $\Box\Diamond\rho$.

3. That $\rho$, once true, remains true unless the server transition occurs, i.e., $\Box(\rho \mathcal{W} t_L)$.

### 7.9.2 Mapping to *OS_reads_F_progress*

As we described above, the conditions $\rho$ that must hold for the *F_reads* transition to map to *OS_reads_F* are as follows:

1. The $F$ mailbox is completely empty, or

2. The only messages in the mailbox are from the *FG* server, or

3. There is at least one message in the mailbox, $F$'s buffer is empty, and the message at the head of the mailbox is not from *FG*.

These conditions are expressed in the following HOL definition:

```
let F_reads_maps_up_if = new_definition(`F_reads_maps_up_if`,
"(∀ (i : num) (trace : trace_def).
  F_reads_maps_up_if i trace =
   let t1 = (trace i)
   and t2 = (trace (SUC i)) in (
     ¬mbx_is_unread_msg
       (filtmbxF_FG
         (get_mbxs_mbx
           (get_trace_mbxs(get_trace_state t1)) F_ID)) ∨
     (mbx_is_unread_msg
       (filtmbxF_FG
         (get_mbxs_mbx
           (get_trace_mbxs(get_trace_state t1)) F_ID)) ∧
      ¬Fs_get_flag(get_trace_Fs(get_trace_state t1)) ∧
      ¬fromF_FG
         (FST
           (read_mbx_msg
             (get_mbxs_mbx
               (get_trace_mbxs(get_trace_state t1))
               F_ID)))))
 )"
);;
```

The function *mbx_is_unread_msg* returns TRUE if there is an unread message in its mailbox argument. The function *filtmbxF_FG* takes a mailbox as a parameter, filters out all of the messages from *F* or *FG*, and returns the rest of the mailbox, in the same order and compressed to eliminate any gaps there might be due to the removal of messages from *F* and *FG*. By checking if there are any unread messages in the filtered *F* mailbox, this definition takes care of the first two conditions in $\rho$.

The function *fromF_FG*, as the name implies, examines a message and returns true if it is from *F* or *FG*. The second disjunctive clause in the definition checks that there is at least one unread message in the mailbox, that the buffer is empty (viz., that the flag is reset), and that the message at the head of the mailbox is from *FG* (or, strictly speaking, *F*, but *F* never sends a message to itself), which is the last of the three cases that make up $\rho$.

The first step in showing that the calling agent system satisfies (via the mapping) *OS_reads_F_progress* is to prove that our $\rho$ is correct. This is done by proving the following theorem:

```
let map_F_reads_conditions = prove_theorem(`map_F_reads_conditions`,
"(∀ (i : num) (trace : trace_def) (F_func G_func : num→num).
  let t1 = (trace i)
  and t2 = (trace (SUC i)) in (
    OS_env_init trace ∧
    OS_env_safety trace ∧
    F_safety trace F_func ∧
    FG_safety trace G_func ∧
    F_reads (get_trace_state t1)(get_trace_state t2) ∧
    F_reads_maps_up_if i trace
  ⟹
    let mt1 = (map_up_element t1)
    and mt2 = (map_up_element t2) in (
      OS_reads_F (get_OStrace_state mt1)(get_OStrace_state mt2))
  )"
);;
```

The function *map_up_element* applies the mapping function to the state in a trace element, returning an OStrace element.

To show that $\rho$ always eventually holds, we use well-founded induction, first finding a termination condition and then proving that it is reached. In this case, $\neg\rho$ implies that there is an unread message in the mailbox that is not from the *FG* agent. The well-founded induction is applied to the distance of this message from the head of the mailbox. The termination condition is when this distance reaches 0, i.e., the message is at the head of the mailbox. We prove that any message in the *F* mailbox eventually advances to the head of the mailbox, so that any message in the mailbox that is not from the *FG* agent will eventually reach the head of the mailbox.

Similarly, we must also prove that if *F*'s buffer is full then it will eventually be emptied (we can do this using the *F_responds_progress* progress property). Together, these results lead to the desired result that, at any point in the trace, either $\rho$ holds or else it eventually holds at some future point in the trace.

To show that $\rho$, once true, remains true until the next *F_reads* transition, we have a problem. $F$ has no control over the other agents, so it cannot ensure that its mailbox will always remain empty until the next *F_reads* transition nor can it guarantee that only the $FG$ agent will send it messages. This means that the first two of the $\rho$ conditions cannot be guaranteed to remain true.

On the other hand, we can prove that each of the other transitions by any agent in $F$'s environment leaves the head of the $F$ mailbox unchanged and leaves the $F$ server buffer empty. This is straightforward (albeit tedious) to prove, as all transitions by other agents (either $FG$ or in the environment) can only append messages to $F$'s mailbox and have no effect on $F$'s local state, and the other transition by $F$ leaves the mailbox unchanged and can only empty the buffer if it is full.

From this we can deduce the following: At any time in a trace when a *F_reads* transition occurs, $F$'s mailbox is either empty, contains only messages from $FG$, or contains at least one message that is not from $FG$. If $F$'s mailbox is empty or contains only messages from $FG$, the *F_reads* transition maps to an *OS_reads_F* transition. If, however, $F$'s mailbox contains at least one message from an agent other than $FG$, we have already proven that a non-$FG$ message will advance to the head of the mailbox and that $F$'s buffer will be empty, and that those conditions will persist until the next *F_reads* transition. Because the *F_reads_progress* progress property guarantees that a *F_reads* transition will happen, we can guarantee that there will always eventually be a time when $\rho$ is true and *F_reads* happens, thus proving that the *OS_reads_F_progress* progress property is preserved.

The complete HOL theorem in which the mapping is proven is as follows:

```
let map_F_reads_progress = prove_theorem(`map_F_reads_progress`,
"(∀ (trace : trace_def) (F_func G_func : num→num).
  OS_env_init trace ∧
  OS_env_safety trace ∧
  F_safety trace F_func ∧
  FG_safety trace G_func ∧
  F_reads_progress trace ∧
  F_responds_progress trace  ==>
  OS_reads_F_progress (map_up_trace trace)
 )"
);;
```

The function *map_up_trace* applies the mapping function to the state in every trace element in the trace, returning an OStrace.

### 7.9.3 Mapping to *OS_responds_F_progress*

The *F_responds* transition is unique among the four transitions in our specifications in that its mapping condition ($\rho$) is always true. There are three possible cases:

1. The *F* server buffer is empty, which maps to an empty *F* calling agent queue. Both transitions are disabled and implement stuttering steps.

2. The *F* calling agent buffer contains a message from *FG*, which maps to an empty *F* entry point queue. Because the result of the *F_responds* transition is an empty buffer that maps to an empty entry point queue, at the entry point level the transition is disabled and it implements a stuttering step.

3. The *F* server buffer contains a message that is not from the *FG* server, which maps to a *F* calling agent queue that contains a single message. The empty buffer after the server transition maps to an empty queue, which is also the result of the calling agent responding to the message.

As a result, a *F_responds* transition always maps to an *OS_responds_F_progress* transition and the proof that the mapping from the calling agent system preserves the progress property *OS_responds_F_progress* in the composed system is extremely simple.

### 7.9.4 Mapping to *OS_reads_FG_progress*

We used virtually the identical method described above for the *OS_reads_F_progress* progress property to prove that that the mapping from the calling agent system preserves the progress property *OS_reads_FG_progress* in the composed system. Other than renaming, the proof is essentially the same.

### 7.9.5 Mapping to *OS_responds_FG_progress*

The following conditions ($\rho$) must hold for the *FG_responds* transition to map to *OS_responds_FG*:

1. The *FG* agent's buffer and queue must both be empty. This maps to an empty queue in the *FG* entry point. If the queue is empty, there are no outstanding response messages expected from the *F* calling agent, so no messages from *F* can be in the mailbox (nor will *FG* ever send itself a message). As a result, the mailbox looks the same to both the *FG* agent and *OSFG* entry point. The *FG_responds* transition implements a stuttering step, which maps to a stuttering step in the *OS_responds_FG* transition.

2. *FG*'s buffer is full and the message is from the *F* calling agent. The message in the buffer from *F* is a response to a request from *FG* that corresponds to the request

message at the head of the *FG* calling agent's request message queue. The request message at the head of *FG*'s queue is the same as the message at the head of the *OSFG* entry point queue. When the *FG_responds* transition sends a response message, that maps to the *OS_responds_FG* transition sending a response message.

These conditions are expressed in the following HOL definition:

```
let F_responds_maps_up_if = new_definition(`F_responds_maps_up_if`,
 "(∀ (trace : trace_def) (i : num).
  F_responds_maps_up_if trace i =
   let t1 = (trace i) in
   let fgi = get_trace_FGs(get_trace_state t1) in
   let fgi_flag = FGS_get_flag fgi
   and fgi_buf = get_FGs_buf fgi
   and fgi_q = FGs_get_queue fgi in (
     ((NULL fgi_q) ∧ ¬fgi_flag) ∨
      (fgi_flag ∧ (from_F fgi_buf)))
 )"
);;
```

As before, we have to prove that this $\rho$ is correct, that it eventually holds, and that once it holds it remains true until a *FG_responds* transition occurs.

The correctness of $\rho$ is proven in the following theorem:

```
let map_FG_reads_conditions = prove_theorem(`map_FG_reads_conditions`,
"(∀ (i : num) (trace : trace_def) (F_func G_func FG_func : num→num).
   let t1 = (trace i)
   and t2 = (trace (SUC i)) in (
     (∀
     OS_env_init trace ∧
     OS_env_safety trace ∧
     F_safety trace F_func ∧
     FG_safety trace G_func ∧
     F_init trace ∧ FG_init trace ∧
     FG_responds (get_trace_state t1)(get_trace_state t2) G_func ∧
     FG_responds_maps_up_if trace i
     ==>
     let mt1 = (map_up_element t1)
     and mt2 = (map_up_element t2) in (
       OS_responds_FG (get_OStrace_state mt1)(get_OStrace_state mt2))))
 )"
);;
```

*F_init* and *FG_init* are included in the antecedents in order to be able to prove a number of invariants. For example, it must be proven that there are never any messages from $F$ in $FG$'s request queue and that there is never a message from $FG$ in $FG$'s buffer or queue.

Another invariant that must be proven to show that $\rho$ is correct is that whenever a response message from $F$ is in $FG$'s buffer and the message at the head of $FG$'s queue has data value $x$, the data value in the message in $FG$'s buffer is equal to $F\_func(G\_func(x))$.

To show that $\rho$ always eventually holds, we again use well-founded induction, first finding a termination condition and then proving that it is reached. In this case, $\neg\rho$ implies that one of the following conditions is true:

1. The $FG$ agent's buffer is empty, but its request queue has at least one message. The $FG$ agent will perform a stuttering step, which maps to a stuttering step even though the *OS_responds_F* transition is enabled to send a response.

2. The $FG$ agent's buffer is full, but the message in the buffer is not from $F$. The $FG$ agent will put the message into the request queue and send a request message to $F$. This does not map to an *OS_responds_F* transition because the request message simply moves from the buffer to the queue, which maps to a stuttering step even though the *OS_responds_F* transition is enabled to send a response.

If the first condition of $\neg\rho$ holds, that means that at some time in the past the $FG$ agent sent a request message to $F$, which will eventually read the message from $FG$ and send a response. The response message from $F$ will make its way to the front of $FG$'s mailbox, and be moved into $FG$'s buffer by a *F_reads* transition. At that time, the condition $\rho$ will hold.

If the second condition of $\neg\rho$ holds, then as a result of the *FG_responds* transition, $FG$ will put the message in its queue and send a request message to $F$. At this point, we are in exactly the same circumstances as if the first condition held: $F$ will eventually read the message and send a response message to $FG$, the message will advance to the head of $FG$'s mailbox and eventually be moved by an *FG_reads* transition into the buffer, at which time the condition $\rho$ will hold.

Our termination condition, then, is when the response message from $F$ finally makes its way into $FG$'s queue. To prove that this termination condition always eventually occurs, we define an abstract queue of messages passed between the $F$ and $FG$ calling agents that is parallel to the $OSFG$ entry point queue. The abstract queue is constructed from the concatenation of the following components, beginning from the head of the abstract queue and working toward the tail:

1. The message in the $FG$ buffer, if it exists and if it is from the $F$ server.
2. All messages in the $FG$ mailbox from the $F$ server, in order.
3. The message in the $F$ buffer, if it exists and if it is from the $FG$ server.
4. All messages in the $F$ mailbox from the $FG$ server, in order.

As an invariant, we prove that the abstract queue is the same length as the $FG$ agent queue. Furthermore, the invariant shows a correspondence between the data values in the abstract queue and the $FG$ queue. For all data values $x$ in the $FG$ queue, the corresponding data values in the messages in the abstract queue for components 1 and 2 are $f(g(x))$ (because they are response messages from $F$). The corresponding data values in the messages in the abstract queue for components 3 and 4 are $g(x)$ (because they are request messages sent by *FG* to *F*). This invariant is proven in the following HOL theorem:

```
let map_FGq_invariant = prove_theorem(`map_FGq_invariant`,
"(∀ (i : num) (trace : trace_def) (F_func G_func FG_func : num→num).
  OS_env_init trace ∧
  OS_env_safety trace ∧
  F_init trace ∧ FG_init trace ∧
  F_safety trace F_func ∧
  FG_safety trace G_func  ⟹ (
  let fgq =
    (FGs_get_queue(get_trace_FGs(get_trace_state(trace i)))) in
  let fglst = (mk_fg_msg_list(get_trace_state(trace i)))
  and glst = (mk_g_msg_list(get_trace_state(trace i))) in
  let par = (APPEND fglst glst) in (
    (LENGTH fgq = LENGTH par) ∧
    (∀x. (0 <= x) ∧ (x < LENGTH fglst) ⟹
      (get_msg_data(EL x par) =
       F_func(G_func(get_msg_data(EL x fgq))))) ∧
    (∀x. (LENGTH fglst <= x) ∧ (x < LENGTH par) ⟹
      (get_msg_data(EL x par) =
       G_func(get_msg_data(EL x fgq)))))
 )"
);;
```

The function *mk_fg_msg_list* constructs the part of the abstract queue that comes from *FG*'s buffer and mailbox. The function *mk_g_msg_list* constructs the part of the abstract queue that comes from the *F* buffer and mailbox. The abstract queue, called *par* for "parallel", is formed from the concatentation of the two parts.

We can use the abstract queue *par* to prove that $\rho$ will always eventually occur. To do this, we use the other progress properties to prove that any message in one of the components of *par* will always eventually advance to the next component. For example, the progress property on the *FG_responds* transition guarantees that a non-*F* message in the *FG* buffer that contains the data value $x$ will eventually cause the *FG* server to send a message to the *F* mailbox that contains the data value $g(x)$. We can use a previously proven result, that any message in the *F* mailbox eventually advances to the head of the mailbox, along with the progress property on the *F_reads* transition, to prove that any message from the

*FG* server in the *F* mailbox eventually advances to the *F* buffer. We use the other progress properties in a similar manner, along with our definition of *par* and the invariants that we proved about it, to show that if $\rho$ does not hold then there always eventually will be a message from the *F* server, containing the value $f(g(x))$, in the *FG* buffer that corresponds to the message at the head of *FG*'s queue, which contains the data value $x$. Thus, at any time either $\rho$ holds or it eventually will.

Proving that $\rho$ persists until the next *FG_responds* transition is a comparatively simpler task. If *FG*'s buffer and queue are empty, there is no guarantee that they will still be empty in the future, but if they are not then we have already proven that the other $\rho$ condition will always eventually hold. No other transition can empty a full *FG* buffer, so once we are in a situation where *FG*'s buffer contains a message from *F*, it will remain there until the next *FG_responds* transition. Together with the progress property on the *FG_responds* transition, we have all necessary theorems and definitions to prove that the *OS_responds_FG_progress* property is preserved by the mapping. The HOL theorem that proves the mapping to *OS_responds_FG_progress* is as follows:

```
let map_FG_responds_progress = prove_theorem(`map_FG_responds_progress`,
"(∀ (trace : trace_def) (F_func G_func FG_func : num→num).
   (∀ (x : num). assumption F_func G_func FG_func x) ∧
   OS_env_init trace ∧
   OS_env_safety trace ∧
   F_init trace ∧ FG_init trace ∧
   F_safety trace F_func ∧
   FG_safety trace G_func ∧
   F_reads_progress trace ∧
   F_responds_progress trace F_func  ∧
   FG_reads_progress trace  ∧
   FG_responds_progress trace G_func ⟹
   OS_responds_FG_progress (map_up_trace trace) FG'func
  )"
);;
```

Note that this theorem uses the proof obligation *assumption* (described in section 7.8) in the antecedents, to ensure that $(F\_func(G\_func\ x) = FG\_func\ x)$.

## 7.10 Applying the Template

In the template, each of the proof steps are saved as theorems. A formal proof about the composition of a system like the $F$ and $FG$ calling agents in our example can be done using the pre-proven theorems by instantiating the universally quantified functions *F_func*, *G_func*, and *FG_func*, and by providing a proof (what we have been calling *assumption*) that $(F\_func(G\_func\ x) = FG\_func\ x)$ for all values in the domain of the three functions.

In many cases, the *assumption* proof can be done almost automatically in HOL. For example, the proof of the composition of some numeric functions can be done in only a few lines using the HOL "arith" library, which implements a partial decision procedure for arithmetic with natural numbers [9]. The complete HOL proof of the theorem that says that $(F\_func(G\_func\ x) = FG\_func\ x)$ in all of our instantiations of the template is as follows:

```
let proveF_FG = prove_thm('proveF_FG',
  "(∀x. (F_func(G_func x) = (FG_func x))
  )",
 GEN_TAC THEN
 REWRITE_TAC [F_func;G_func;FG_func] THEN
 CONV_TAC ARITH_CONV
);;
```

Given the proof about the composition of the server functions, the rest of the proof is similarly simplified by using the pre-proven theorems and several proof tactics, which apply the theorems to the composition proof goals.

The theorems and definitions constitute a HOL theory about composing two simple servers. This theory serves as a template for all compositions of similar servers. By simply specifying the functions calculated by the servers, the composition of the servers can to a great degree be done automatically.

We did this for our example template, choosing two different sets of functions. We used the same $G\_func(x) = x + 2$ for both instances. In the first instance, we chose $F\_func(x) = x+1$ and $FG\_func(x) = x+3$. In the second instance, we chose $F\_func(x) = x * 2$ and $FG\_func(x) = 2x + 4$. No other substitutions were made. These two instances of the template are shown in appendix B.

## 7.11 Generalizing the Template

As the template is currently written, although it can be reused for different values of *F_func*, *G_func*, and *FG_func*, it is otherwise not particularly general and is not entirely suitable for reuse in the incremental composition of a hierarchy of calling agents. Here are some modifications that would generalize the template and make it more useful for reuse in a hierarchical composition:

- Use Abadi and Lamport's method of specifying local state. Our method of specifying local state, described in section 4.3, puts the local state in the same data type as the global state. This makes incremental composition difficult to automate, because the local state for every component that is being composed must be "manually" added to the trace state definition for each composition. Ideally, each component specification should be the only place where the component's local state is mentioned. Using Abadi and Lamport's method of specifying local state we could also use the same trace type definition for both the agents and the entry points, rather than the two that we used in the template.

- Use the same types of local state for the calling agents as for the entry points. In this template, the entry points use our standard calling agent model (see section 5.1), but the calling agents do not: instead of a queue of messages they have only a single-length buffer.

  Note that would be impossible in our model for the *OSFG* entry point to also have a single-length buffer because the *FG* agent must queue request messages until the *F* agent responds. No mapping would be possible unless the *OSFG* agent also queued messages, which is why a queue is built into our standard calling agent model. Using the standard model in our template would make it possible to apply the template to compose calling agents that are themselves compositions of other agents.

- Convert the identifiers from an enumerated type to a set. As defined in this template, the identifiers *SYS_F*, *SYS_FG*, and *ENV* are hard-coded (see section 7.2.3). That is sufficient for this particular template, because the result of the composition is not itself being composed with anything else. If it were to be composed with another specification, however, there would be no way to indicate that the identifiers in the other system are part of this system's environment. The identifiers in this system must be in the form of a set rather than an enumerated type. Then it would still be straightforward to identify the system agents and the environment agents (viz., anything of the agent type that is not in the set of system agents), and would leave the identity of the environment agents unspecified until it is necessary to explicitly name them.

- Parameterize the identifiers. This template works great for things that are labeled $F$ and $FG$, but not everything in a hierarchy can have the same names. We should be able to specify *SYS_F* and *SYS_FG*, or $A$ and $B$, or any other identifiers of the proper type (as defined for the set of identifiers; see the bulleted item above) that we choose.

## 7.12 Conclusion

In this chapter, we have described the specifications and proofs of an example template. The template demonstrates our refinement of Abadi and Lamport's method, our basic model of calling agents, and our method of using reusable composition proofs, where functions (e.g., *F_func*) and proofs (e.g., *assumption*) are parameters to the composition proof. The template also demonstrates how two agents can be composed into a compound agent.

Note that our composition method does not necessarily give any savings with respect to the work involved in creating the specifications and proofs in a template (although our standardized specification method might provide some benefit). This work would have to be done regardless of whether one is using templates or doing a compositional proof in some other way. There can be a major reduction in proof effort, however, when templates are reused: not only does our incremental proof strategy reduce the complexity of the proofs in templates (relative to the proof complexity of composing the entire system at once), but reusing a template means that there is almost no additional specification and proof effort required for additional composition stages.

# Chapter 8

# Summary

Beware of bugs in the above code; I have only proved it correct, not tried it.
*Donald Knuth, March 1977 memo to Peter van Emde Boas, cited on* `http://Sunburn.Stanford.EDU/~knuth/faq.html`

Computers have been integrated into many different types of systems where the safety of lives and property directly depends on the correctness of the system. To have the maximum assurance that a system is correct, a system should be formally verified. It can be quite difficult, however, to formally verify large, complex systems. As the size and complexity of the system increases, so can the size and complexity of the verification effort.

Using the "divide-and-conquer" technique of compositional proof, a system's components can be independently verified, and then the verified components "composed" into the complete system. Compositional proof can significantly reduce the complexity of verifying a system; once a component is independently verified its internal details are hidden when the component is composed with other system components. Moreover, the specification and proof of correctness for a component can be reused when the component is reused.

Compositional proofs themselves, however, can be quite difficult. While the complexity of verifying a system using compositional methods may be many orders of magnitude less than verifying the system as a monolithic unit, it may still be too great to be practical. Compositional proof hides the internal details of components (from the point of view of the other components), but there may be many components and the compositional proof must deal with interactions between all of the external details of all the components. Thus, even compositional proof may prove to be an intractable exercise for real systems, where the time and cost to carry out the verification determine whether or not it will be done at all.

Another hurdle in applying compositional proof is that, in general, each proof is "hand-made" — constructed for the verification of a specific system and tailored to that system alone. Ideally, the work put into the compositional verification of a component with other components in a system could be reused along with the component's specification and proof of correctness, and would require a minimum of "customization."

The goal of the work presented here is to make the compositional proof of large, complex systems tractable. Our approach is to simplify composition by incrementally composing parts of the system in a step-wise fashion. This reduces the complexity of each step of the proof.

We also have designed a standard model of components and templates, which consist of abstract specifications and generic proofs, that can be reused to compose a wide variety of components by instantiating different parameters. Using a small number of templates, the composition of a large system can be performed with a bare minimum, or even no, additional proof effort.

## 8.1 Major Contributions

The result of our work is a practical methodology for the composition of large, distributed systems — a scalable methodology that enables the reuse of components, specifications, and compositional proofs. The work is applicable to a wide variety of systems that are either distributed or where the system policies are distributed among distinct components, including micro-kernel operating systems [19], the DNS, web caching systems, layered security policies [18], and others. The major contributions of our work are as follows:

1. Our calling agent model is a simple, yet easily extensible model of components in a distributed system.

2. Our specification style constraints on Abadi and Lamport's general composition model provide a framework for the development of compositional templates. The constraints ensure that abstract specifications for calling agents satisfy the requirements of the composition rule, and guide the generic composition proofs.

3. Our notion of templates changes the formal verification of large systems from an *ad hoc* activity into an engineering exercise that can be applied using well-defined rules.

4. Our calling agent hierarchy abstraction and incremental composition algorithm can be used to find an optimal incremental composition strategy. An optimal strategy reduces the overall complexity of the compositional proof to a minimum.

5. We have provided a fully worked example of a template, including generic specifications for calling agents and a generic composition proof.

Through the use of templates and incremental composition, the effort that goes into the compositional proof of a large, complex system can be reduced to the point of simply choosing suitable parameters and supplying them to a theorem-prover.

## 8.2 Future Work

The work presented here has shown how templates and iterative composition can be used to reduce compositional proof complexity in distributed systems and systems whose policies are distributed among a number of components. The success of this effort encourages us to continue research in the following areas:

- **Further develop and refine templates and their HOL implementation.** Our demonstration as we presented it is still unsuitable for use as an engineering tool by unsophisticated users. A tool can be created that uses templates but that hides the specification and proof details. This tool will permit engineers to verify the correctness of their system design simply by providing appropriate parameters, even without understanding the intricacies of compositional proof.

- **Support for n-ary trees.** Our method focuses on tree-like topologies, but it does not yet support, in an elegant way, trees of arbitrary degree. Either "identity" agents must be used to fill out subtrees of lower degree, or else different templates must be used for subtrees of different degrees. We would like to make a more generic template in which the degree of the subtree is parameterized.

- **Additional topologies.** Our basic model of a calling agent requires a response message to every request message. A more general model would permit requests that do not generate responses, or message "forwarding" so that responses can come from a different agent than the request message recipient.

- **Automatic synthesis of code and proof from specifications.** Only the arguments to template parameters must be supplied to verify the composition of the calling agents that are instantiations of the templates. In effect, the specifications and composition proofs for the calling agents are being synthesized from the parameters. The calling agent model is simple and regular, so it may also be possible to synthesize the actual implementation (viz., code) for the calling agents. And if we can synthesize code, it may be possible to synthesize the correctness proofs for the calling agents — i.e., that the synthesized code satisfies the synthesized specifications. Thus, by supplying appropriate template parameters, an engineer conceivably could synthesize an entire, verified distributed system.

## 8.3 Conclusion

The goal of the work presented here is to reduce the complexity of compositional proof to the level of an engineering exercise that can be applied in a regular fashion, following well-defined rules, and using well-understood tools. The standard model of a calling agent and the standard specifications and proofs that make up a template are a means toward this end. By using templates along with incremental composition, the compositional proof effort to verify a system can be reduced by many orders of magnitude, and provide an environment in which compositional proof can be carried out by users who have little experience with formal verification. The techniques that we have developed point the way toward a method of integrating formal verification as a practical step in the development of large, complex systems.

# Appendix A

# Template Specifications

This appendix contains the complete specification files for the example template described in chapter 7. The files are written in the HOL88 language (basically ML, with extensions). Readers may wish to consult reference [16] or the following web pages for more information about HOL:

- http://www.cl.cam.ac.uk/Research/HVG/HOL/
- http://lal.cs.byu.edu/lal/hol-documentation.html

The abstract specification for the template was developed using an earlier specification created as part of the UCD Silo Project [39], which is why many of the file names begin with the prefix "silo".

The file init.ml contains some basic HOL "bookkeeping" commands and some simple tactics that are used in the proof.

The files silomorelists.ml, silolists.ml, and silomappings.ml contain basic definitions and theorems for lists and for a VDM-like mapping type [21]. The file silobasic.ml contains the definitions for messages, mailboxes, and other global and local state types. Each of these files contains constructor and destructor functions for the types that it defines.

The F-FG.ml file contains the definitions of the two calling agents. The OS-env.ml file contains the system environment definitions, while the OS-sys.ml file contains the definitions for the composed system. The prefix "OS" means "overall system" in this context.

The letconv.ml and siloprojection.ml files contain basic definitions and theorems used by the mapdefs.ml file. The mapdefs.ml file contains the mapping function used in the refinement step.

## A.1 init.ml

```
%****************************** -*- Mode: Hol -*- *******************
** init.ml --- Standard miscellaneous definitions.
*
* Mark Heckman  10/04/95
*******************************************************************%

%-----------------------------------------------------------------
                          Turn off prompt
-----------------------------------------------------------------%

message(`Turning off prompt.`);; set_flag(`prompt`, false);;

%-----------------------------------------------------------------
                          Turn on timing
-----------------------------------------------------------------%

message(`Turning on timing.`);; set_flag(`timing`, true);;

%-----------------------------------------------------------------
                           autoload_all

Autoload all definitions and theorems from the given THEORY. The
call will fail if THEORY is not part of the current theory
```

```
segment.

Acquired from Rob Shaw <shaw@cs.ucdavis.edu>, who acquired it from
Phil Windley and Mike Gordon. Was named my_autoload_theory.
-------------------------------------------------------------------%

let autoload_all thy =
    map (\name. autoload_theory(`axiom`, thy, name))
        (map fst (axioms thy));
    map (\name. autoload_theory(`definition`, thy, name))
        (map fst (definitions thy));
    map (\name. autoload_theory(`theorem`, thy, name))
        (map fst (theorems thy));
    ();;

%-------------------------------------------------------------------
                             load_parent

Declare a new PARENT theory and autoload all definitions and
theorems from that theory.

Acquired from Rob Shaw <shaw@cs.ucdavis.edu>, who acquired it from
Phil Windley and Mike Gordon.
-------------------------------------------------------------------%

let load_parent parent =
    new_parent parent;
    autoload_all parent;;

%-------------------------------------------------------------------
                           new_theory_safe

Start a new THEORY definition. If a file THEORY.th exists on the
search_path, the first such file will be deleted.

BEWARE! I believe that this does what I want it to do, but you use
it at your own risk!

It still has problems if THEORY has been previously opened during
the current session. I do not at the moment know how to fix this.
-------------------------------------------------------------------%

let new_theory_safe theory =
    (
    let theory_file_path = find_file (theory ^ `.th`)
    in
        message(`Removing file:  ` ^ theory_file_path);
        unlink theory_file_path
    ) ?? [`find_file`] ();
    new_theory theory;
    message(`Theory ` ^ theory ^ ` opened.`);;

%<*********************************************************************
                              TACTICS, ETC.
*********************************************************************>%

%<=====================================================================
                                    c

Like e, the alias for expand, but uses CHANGED_TAC so that failure
occurs if nothing changes in the goal.
=====================================================================>%

let c x = e(CHANGED_TAC x);;
%let e x=expand(CHANGED_TAC x);;%

let SYM_RULE =
  (CONV_RULE (ONCE_DEPTH_CONV SYM_CONV))
? failwith `SYM_RULE`;;

let EXPAND_LET_TAC =
   ONCE_REWRITE_TAC [LET_DEF]
   THEN CONV_TAC (TOP_DEPTH_CONV BETA_CONV);;

let EXPAND_LET_RULE x =
   (CONV_RULE (TOP_DEPTH_CONV BETA_CONV)
      (ONCE_REWRITE_RULE [LET_DEF] x));;

let REMOVE_DUP_ASM_TAC = (
  POP_ASSUM_LIST(\thl. MAP_EVERY ASSUME_TAC (setify thl))
);; let UNDISCH_EVERY_TAC = (
  ASSUM_LIST (\thl.
    EVERY (map (UNDISCH_TAC o concl) (setify thl))
  )
);; let UNDISCH_FIRST_TAC = (
  ASSUM_LIST (\thl. (UNDISCH_TAC o concl) (el 1 thl))
);; let POP_ALL = (POP_ASSUM_LIST(\thl. ALL_TAC));; let
POP_ALL_TAC = (POP_ASSUM_LIST(\thl. ALL_TAC));;
```

```
let let_CONV_RULE = (CONV_RULE (DEPTH_CONV let_CONV));; let
let_CONV_TAC = (CONV_TAC (DEPTH_CONV let_CONV));; let
sym_ASM_EL_TAC n = (ASSUM_LIST(\thl.
                   ASSUME_TAC (SYM_RULE (el n thl)))
);; let ASM_REWRITE_EL_TAC n = (ASSUM_LIST(\thl.
                   REWRITE_TAC [(el n thl)])
);; let sym_ASM_REWRITE_EL_TAC n = (ASSUM_LIST(\thl.
                   REWRITE_TAC [SYM_RULE (el n thl)])
);; let PROMOTE_ASM_TAC s = (
  (UNDISCH_TAC s THEN
  STRIP_TAC) ? failwith `PROMOTE_ASM_TAC`
);; let ASM_REWRITE_PICK_TAC al thl = (
  ASSUM_LIST(\asl.
    REWRITE_TAC ((filter ((\a. mem a al) o concl) asl) @ thl)
  )
);; let ALL_THM_TAC th = ALL_TAC;; let TRASH_ASSUM as =
  PROMOTE_ASM_TAC as THEN POP_ASSUM ALL_THM_TAC
  ? failwith `TRASH_ASSUM: assumption not found`;;
let filter_as_list asl al = (
  (filter ((\a. mem a al) o concl) asl)
);; let KEEP_ASM_TAC al =
  POP_ASSUM_LIST(\asl.
    EVERY (map ASSUME_TAC (filter_as_list asl al))
  )
;; let REWRITE_ASM_PICK_TAC as thl al = (
  ASSUM_LIST(\asl.
    if not ((filter ((\a. a = as) o concl) asl) = [])
    then ASSUME_TAC (
      REWRITE_RULE (filter_as_list asl al @ thl)
                   (find ((\a. a = as) o concl) asl)
     ) THEN TRASH_ASSUM as
    else failwith `REWRITE_ASM_PICK_TAC`
  )
);; let sym_ASM_PICK_TAC as = (
  ASSUM_LIST(\asl.
    if not ((filter ((\a. a = as) o concl) asl) = [])
    then ASSUME_TAC (
      SYM_RULE (find ((\a. a = as) o concl) asl)
    ) THEN TRASH_ASSUM as
    else failwith `sym_ASM_PICK_TAC`
  )
);; let mod_ASM_PICK_TAC as rl = (
  ASSUM_LIST(\asl.
    if not ((filter ((\a. a = as) o concl) asl) = [])
    then ASSUME_TAC (
      rl (find ((\a. a = as) o concl) asl)
    ) THEN TRASH_ASSUM as
    else failwith `mod_ASM_PICK_TAC`
  )
);;

let REWRITE_ASM_KEEP_PICK_TAC as thl al = (
  ASSUM_LIST(\asl.
    if not ((filter ((\a. a = as) o concl) asl) = [])
    then ASSUME_TAC (
      REWRITE_RULE (filter_as_list asl al @ thl)
                   (find ((\a. a = as) o concl) asl)
     )
    else failwith `REWRITE_ASM_KEEP_PICK_TAC`
  )
);; let ONCE_REWRITE_ASM_PICK_TAC as thl al = (
  ASSUM_LIST(\asl.
    if not ((filter ((\a. a = as) o concl) asl) = [])
    then ASSUME_TAC (
      ONCE_REWRITE_RULE (filter_as_list asl al @ thl)
                   (find ((\a. a = as) o concl) asl)
     ) THEN TRASH_ASSUM as
    else failwith `ONCE_REWRITE_ASM_PICK_TAC`
  )
);; let sym_ASM_KEEP_PICK_TAC as = (
  ASSUM_LIST(\asl.
    if not ((filter ((\a. a = as) o concl) asl) = [])
    then ASSUME_TAC (
      SYM_RULE (find ((\a. a = as) o concl) asl)
    )
    else failwith `sym_ASM_KEEP_PICK_TAC`
  )
);; let mod_ASM_KEEP_PICK_TAC as rl = (
  ASSUM_LIST(\asl.
    if not ((filter ((\a. a = as) o concl) asl) = [])
    then ASSUME_TAC (
      rl (find ((\a. a = as) o concl) asl)
    )
    else failwith `mod_ASM_PICK_TAC`
  )
);;
```

```
let list_mp_ASM_PICK_TAC al as = (
  ASSUM_LIST(\asl.
    if not ((filter ((\a. a = as) o concl) asl) = [])
    then ASSUME_TAC (
      LIST_MP (filter_as_list asl al)
              (find ((\a. a = as) o concl) asl)
    )
    else failwith `list_mp_ASM_PICK_TAC`
  )
);; let match_mp_ASM_PICK_TAC as1 as2 = (
  ASSUM_LIST(\asl.
    if not ((filter ((\a. a = as1) o concl) asl) = [])
    then (if not ((filter ((\a. a = as2) o concl) asl) = [])
    then ASSUME_TAC (
      MATCH_MP (find ((\a. a = as2) o concl) asl)
              (find ((\a. a = as1) o concl) asl)
    )
    else failwith `list_mp_ASM_PICK_TAC--can not find 2nd assumption`
    )
    else failwith `list_mp_ASM_PICK_TAC--can not find 1st assumption`
  )
);;
```

## A.2 silomorelists.ml

```
%-------------------------------------
* File:    silomorelists.ml
* Version: 0.0
* Date:    03/31/96
*-------------------------------------%

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Definitions and theorems for lists.

To maintain compatibility with code that uses the old "more-lists"
library, this theory contains several definitions, such as
"MEMBER", that are not in the 2.02 lists library.
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

loadf `aux/init.ml`;;
```

```
new_theory_safe `silomorelists`;;

load_parent `list`;;

%=================================================================
      begin -- Borrowed or adapted from
               the HOL-2.01 more_lists library.
=================================================================%

let MEMBER =
    new_list_rec_definition(`MEMBER`, "
        (MEMBER (x : *) [] = F) /\
        (MEMBER x (CONS h t) = (x = h) \/ MEMBER x t)
    ");;

let MEMBER_SNOC = prove_thm (`MEMBER_SNOC`,
 "!t (x:*) h. MEMBER x (SNOC h t) = (h = x) \/ MEMBER x t",
 LIST_INDUCT_TAC THEN
 ASM_REWRITE_TAC[MEMBER;SNOC] THENL [
   REPEAT GEN_TAC THEN
   GEN_REWRITE_TAC
     (RAND_CONV o ONCE_DEPTH_CONV) [] [EQ_SYM_EQ] THEN
   REWRITE_TAC []
 ;
   REWRITE_TAC [DISJ_ASSOC] THEN
   REWRITE_TAC
     [(SPECL ["((x:*)=h)";"(h'=(x:*))"] DISJ_SYM)]
 ]
);;

let SNOC_NOT_NULL = prove_thm(`SNOC_NOT_NULL`,
  "! (d:*) l. ~NULL (SNOC d l)",
 REPEAT GEN_TAC THEN
 STRUCT_CASES_TAC (SPEC "l:(*)list" list_CASES) THEN
 REWRITE_TAC[SNOC;NULL]
);;

let NULL_EQ_EMPTY = prove_thm (`NULL_EQ_EMPTY`,
 "!l:(*)list .  NULL l = (l = [])",
 GEN_TAC THEN
 STRUCT_CASES_TAC (SPEC_ALL list_CASES) THEN
 REWRITE_TAC [NULL;NOT_CONS_NIL]
);;
```

```
let SNOC_APPEND_CONS = prove_thm(`SNOC_APPEND_CONS`,
   "!(l:(*) list) h. SNOC h l = APPEND l [h]",
   LIST_INDUCT_TAC THENL
   [ REWRITE_TAC [SNOC;APPEND];
     ASM_REWRITE_TAC [SNOC;APPEND]]
);;


let HD_SNOC = prove_thm(`HD_SNOC`,
  "!(h:*) t. HD (SNOC h t) = NULL t => h | HD t",
 REPEAT GEN_TAC THEN
 STRUCT_CASES_TAC (SPEC "t:(*)list" list_CASES) THEN
 REWRITE_TAC [NULL;SNOC;HD]
);;

let LENGTH_NOT_NULL =
 prove_thm
  (`LENGTH_NOT_NULL`,
   "!(l:(*)list). (0 < LENGTH l) = (~(NULL l))",
   LIST_INDUCT_TAC THENL
   [REWRITE_TAC [LENGTH;NULL;NOT_LESS_0];
    REWRITE_TAC [LENGTH;NULL;LESS_0]]
);;



let LAST1 = prove_thm(`LAST1`, "!l y. LAST (APPEND l [y:*]) = y",
 (REWRITE_TAC[GSYM SNOC_APPEND_CONS;LAST])
);;



let SNOC_LENGTH = TAC_PROOF (([], "! (d:*) l. LENGTH (SNOC d l) =
SUC (LENGTH l) "),
 GEN_TAC THEN
 LIST_INDUCT_TAC THEN
 ASM_REWRITE_TAC [LENGTH;SNOC]);;


let LLESS_LEFT = new_infix_list_rec_definition(`LLESS_LEFT`,
      "(LLESS_LEFT ([]:* list) l = ~(NULL l)) /\
       (LLESS_LEFT (CONS a rest)  l =
             ~(NULL l) /\ (a = HD l) /\ (LLESS_LEFT rest (TL
l)))");;

let LLEQ_LEFT = new_infix_definition(`LLEQ_LEFT`,
      "LLEQ_LEFT (l1:* list) l2 = (l1 LLESS_LEFT l2) \/ (l1 =
l2)");;

%   *************************************************************
    *                                                           *
    *    Give a meaning to the tail of an empty list.           *
    *                                                           *
    *    TAIL = |- (TAIL[] = []) /\ (!h t. TAIL(CONS h t) = t)  *
    *                                                           *
    *************************************************************%


let TAIL = new_recursive_definition false list_Axiom `TAIL`
       "(TAIL ([]:* list) = ([]:* list)) /\
        (TAIL (CONS (h:*) t) = t)";;



let NTAIL= new_prim_rec_definition(
  `NTAIL`,
  "(NTAIL 0 (l:* list) = l) /\
   (NTAIL (SUC n) l = TAIL (NTAIL n l) ) ");;

let TL_TAIL = prove_thm(`TL_TAIL`, "!l:* list. ~ (NULL l) ==> (TL
l = TAIL l)",
 LIST_INDUCT_TAC
THENL [ (REWRITE_TAC [NULL]);
  (REWRITE_TAC[TL;TAIL]) ]
);;


%************************************************************
*
*    Define the subsequence of a list as the first n elements.
*
*    SUBSEQ =
*    |- (!x. SUBSEQ 0 x = []) /\
*    (!n x. SUBSEQ (SUC n) [] = []) /\
*    (!n x. SUBSEQ (SUC n ) CONS(HD x)(TAIL x) =
*                       CONS(HD x)(SUBSEQ n(TAIL x)))
*
*****************************************************************%

let SUBSEQ = new_prim_rec_definition (`SUBSEQ`, "(SUBSEQ 0 (x:*
list) = ([]:* list)) /\
 (SUBSEQ (SUC (n:num)) (x:* list) =
         NULL x => [] | CONS (HD x) (SUBSEQ n (TAIL x)))");;

let SUBSEQ_EMPTY_LIST = prove_thm(`SUBSEQ_EMPTY_LIST`,
     "! (n:num). SUBSEQ n ([]:* list) = []",
  (INDUCT_TAC) THEN
  (REWRITE_TAC [SUBSEQ;NULL_EQ_EMPTY]));;

%****************************************************************
*
*    If two lists are equal then equal length subsequences will
*    also be equal.
*
*    SUBSEQ_EQ = |- !x y n. (x = y) ==> (SUBSEQ n x = SUBSEQ n y)
*
*****************************************************************%


let SUBSEQ_EQ = prove_thm (`SUBSEQ_EQ`, "! (x:* list) (y:* list)
(n:num). (x = y) ==>
    ((SUBSEQ n x) = (SUBSEQ n y))",
REPEAT GEN_TAC THEN DISCH_TAC THEN ASM_REWRITE_TAC []);;



let SUBSEQ2 = prove_thm (`SUBSEQ2`, "! (n:num) (x:*) (y:* list).
    CONS x (SUBSEQ n y) = SUBSEQ (SUC n) (CONS x y)",
  INDUCT_TAC THEN
  REWRITE_TAC [SUBSEQ; HD; CONJUNCT2 TAIL;
               NULL_EQ_EMPTY;NOT_CONS_NIL]);;


let SUBSEQ_SUBSEQ = prove_thm (`SUBSEQ_SUBSEQ`,
    "! (n:num) (x:* list). SUBSEQ n (SUBSEQ n x) = SUBSEQ n x",
INDUCT_TAC THENL [
 (REWRITE_TAC [SUBSEQ]);
 (REPEAT GEN_TAC) THEN
 (REWRITE_TAC [SUBSEQ]) THEN
 (ASM_CASES_TAC "NULL (x:* list)") THEN
 (ASM_REWRITE_TAC[NULL;HD;TAIL])
]);;


let HD_APPEND_NULL =
    prove_thm
       ('HD_APPEND',
        "! l1:(*)list. (~(NULL l1)) ==>
            (! l2. (HD (APPEND l1 l2)) = (HD l1))",
        REPEAT STRIP_TAC THEN
        IMP_RES_THEN (ASSUME_TAC o SYM) CONS THEN
        ONCE_ASM_REWRITE_TAC [] THEN
        REWRITE_TAC [APPEND; HD;]
);;


let LENGTH_TAIL =prove_thm( 'LENGTH_TAIL',
 "!l:* list. ~NULL l ==> (LENGTH(TAIL l) = ((LENGTH l) - 1))",
  LIST_INDUCT_TAC THENL
  [ ASM_REWRITE_TAC[NULL] ;
    ASM_REWRITE_TAC[TAIL;LENGTH;SYM(SPEC_ALL PRE_SUB1);PRE]
  ]
);;


let LENGTH_TL = prove_thm('LENGTH_TL',
  "(!l:* list. ~NULL l ==> (LENGTH(TL l) = (PRE(LENGTH l)))
  )",
  REPEAT STRIP_TAC THEN
  IMP_RES_TAC (TL_TAIL) THEN
  UNDISCH_TAC "~NULL (l:(*)list)" THEN
  ASM_REWRITE_TAC [PRE_SUB1;LENGTH_TAIL]
);;



%================================================================
      end -- from the HOL-2.01 more_lists library.
================================================================%


close_theory ();;
```

## A.3 silolists.ml

```
%-------------------------------------
* File:    silolists.ml
* Version: 0.0
* Date:    10/08/96
*--------------------------------------%

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Definitions and theorems for lists.

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

loadf `aux/init.ml`;;

new_theory_safe `silolists`;;

load_parent `list`;; load_parent `silomorelists`;;


% ---------------------------------------------------------------
* INDEX

Returns the index of an element of a list that matches a specified
element.

UNDEFINED IF ELEMENT NOT PRESENT OR IF LIST EMPTY. Safe when used
as in the following example:

      (! elem :: (MEMBER elem lst) .
        ((INDEX elem lst) < some_value)
      )

Where existence of the element in the list is first checked by
MEMBER.

---------------------------------------------------------------- %

let INDEX = new_definition (`INDEX`,"
  (! (el: *) (s : * list) .
    INDEX el s = (
      @x. (EL x s) = el
    )
  )
");;
```

```
% ---------------------------------------------------------------
* extract_substring

Given a string, a starting index and a length, returns the
substring of that length beginning at the index.

Returns the empty string if the starting index plus the length is
greater than the length of the string.

--------------------------------------------------------------- %

let extract_substring = new_definition (`extract_substring`, "
  (! (s : (*)list)  i lngth .
  extract_substring s i lngth =
    (SUBSEQ lngth (NTAIL i s))
  )
");;

% ---------------------------------------------------------------
* is_subseq_general

TRUE if the first list is a subsequence of the second.

i.e., given some lists [a1;...;an] and [b1;...;bm],
is_subseq_general is TRUE if list a is empty or if there is some j
such that j+n<=m and [a1;...;an] equals [b(j+1);...;b(j+n)].

--------------------------------------------------------------- %

let is_subseq_general = new_definition (`is_subseq_general`, "
  (! (subs) (s) : (**)list .
  is_subseq_general subs s =
    (? j .
      ((j + (LENGTH subs)) <= (LENGTH s))
      /\ (subs = (extract_substring s j (LENGTH subs)))
    )
  )
");;


% ---------------------------------------------------------------
* is_prefix
```

```
True if the first list is a prefix of the second.

---------------------------------------------------------------- %

let is_prefix = new_definition (`is_prefix`, "
  (! (subs) (s) : (**)list .
  is_prefix subs s =
    ((LENGTH subs) <= (LENGTH s))
    /\ (subs = (extract_substring s 0 (LENGTH subs)))
  )
");;


% ------------------------------------------------------------------
* newBUTFIRSTN

Similar to BUTFIRSTN, but defined when the list is empty, in which
case the result is the empty list (i.e., if the number of elements
to skip is longer than the list, the result is the empty list).
---------------------------------------------------------------- %

let newBUTFIRSTN = new_prim_rec_definition(`newBUTFIRSTN`,
  "(!(l:(*)list). newBUTFIRSTN 0 l = l) /\
   (!n (l:(*)list).
     newBUTFIRSTN(SUC n)l = (newBUTFIRSTN n (TAIL l)))"
);;



close_theory ();;
```

## A.4 silomappings.ml

```
%-------------------------------------
* File:    silomappings.ml
* Version: 0.0
* Date:    08/12/95
*-------------------------------------%

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Definitions and theorems for Z-like partial functions.
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %
```

```
loadf `aux/init.ml`;;

new_theory_safe `silomappings`;;

load_library `sets`;;


% -------------------------------------------------------------------
* dom
The domain of a partial function.
------------------------------------------------------------------- %
let dom = new_definition (`dom`, "
  (! (map : (*#**)set).
    dom map = (
      {x | ?y. (x,y) IN map}
    )
  )
");;


% -------------------------------------------------------------------
* rng
The rng of a partial function.
------------------------------------------------------------------- %
let rng = new_definition (`rng`, "
  (! (map : (*#**)set).
    rng map = (
      {y | ?x. (x,y) IN map}
    )
  )
");;


% -------------------------------------------------------------------
* map_rng
The rng of a partial function value. If the input set is a
mapping, the result should be a set of no more than one element
(zero, if the specified domain value is not in the domain).
------------------------------------------------------------------- %
let map_rng = new_definition (`map_rng`, "
  (! (map : (*#**)set) (d:*) .
    map_rng map d = (
      {y | (d,y) IN map}
    )
  )
");;


% ----------------------------------------------------------------
* is_mapping
*
A predicate that determines if a set of (*,**) pairs is a mapping.
A mapping has no duplicates of values in its domain.
-------------------------------------------------------------- %

let is_mapping = new_definition (`is_mapping`, "
  (! (map : (*#**)set).
    is_mapping map = (
      !x . (x IN (dom map) ==> SING(map_rng map x))
    )
  )
");;


% ----------------------------------------------------------------
* map_to_extract
Extracts the value from a set containing a single value. Used to
extract a partial function value from the set returned by map_rng.
UNDEFINED IF SET SIZE > 1 OR SET SIZE = 0 (i.e. if source set not
a mapping or if there is no mapping).
-------------------------------------------------------------- %
let map_to_extract = new_definition (`map_to_extract`, "
  (! (r : (**)set).
    map_to_extract r = (
      ((SING r) => CHOICE r | ARB)
    )
  )
");;


% ----------------------------------------------------------------
* maps_to
*
Given a value in the domain of a mapping, returns the value it
maps to in the range.
*
UNDEFINED WHEN INPUT VALUE NOT IN THE DOMAIN OR MULTIPLY DEFINED
```

```
(i.e., it is not a mapping).

To use safely, must use with is_mapping and dom, as in

  (is_mapping m) /\ (x IN (dom m)) => maps_to m x
---------------------------------------------------------------- %

let maps_to = new_definition (`maps_to`, "
  (! (map : (*#**)set) (x : *).
    maps_to map x = (
      map_to_extract (map_rng map x)

    )
  )
");;


% ----------------------------------------------------------------
* map_overwrite

Replaces all pairs in one set with pairs from a second set whose
first elements match. Preserves pairs from the first set for which
there are not matching elements in the second set. Adds pairs in
the second set whose first element do not match any in the first
set. Essentially, this function returns the union of two mappings,
selecting the pair from the second set when a conflict occurs.
Like map_replace, but this function can increase the size of the
domain of a mapping.
---------------------------------------------------------------- %

let map_overwrite = new_definition (`map_overwrite`, "
  (! (m1 m2 : (*#**)set) .
    map_overwrite m1 m2 = (
      {(x,y) |
        ((x,y) IN m2) \/
        (((x,y) IN m1) /\ ~(x IN dom m2))
      }
    )
  )
");;

% ----------------------------------------------------------------
* map_replace
```

```
Replaces all pairs in one set with pairs from a second set whose
first elements match. Preserves pairs from the first set for which
there are not matching elements in the second set. Discards pairs
in the second set whose first element do not match any in the
first set. Like map_overwrite, but this function preserves the
domain of a mapping.
---------------------------------------------------------------- %

let map_replace = new_definition (`map_replace`, "
  (! (m1 m2 : (*#**)set) .
    map_replace m1 m2 = (
      {(x,y) |
        (((x,y) IN m2) /\ (x IN dom m1)) \/
        (((x,y) IN m1) /\ ~(x IN dom m2))
      }
    )
  )
");;

% ----------------------------------------------------------------
* map_remove

Removes all pairs in one set whose first elements match elements
of the second set. Preserves pairs from the first set for which
there are not matching elements in the second set. Ignores
elements in the second set whose do not match any first elements
in the first set. This function reduces the size of the domain of
a mapping.
---------------------------------------------------------------- %

let map_remove = new_definition (`map_remove`, "
  (! (m1 : (*#**)set) (m2 : (*)set) .
    map_remove m1 m2 = (
      {(x,y) |
        ((x,y) IN m1) /\ ~(x IN m2)
      }
    )
  )
");;

close_theory ();;
```

## A.5 silobasic.ml

```
%--------------------------------------
* File:    silobasic.ml
* Version: 0.0
* Date:    10/04/95
*---------------------------------------%

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
The basic types and other definitions used in the silo OS
specification.
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

loadt `aux/init.ml`;;

new_theory_safe `silobasic`;;

load_library `sets`;;

load_parent `silomappings`;;

load_parent `silolists`;;


% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define the local and global system state types
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% =================================================================
*
*       basic state types
*
* ================================================================%

new_type_abbrev (`task_id`, ":num");;

% =================================================================
*
*       local state
*

The local state for each task consists of the registers and pc.

Local state is implemented as a partial function from task_ids to
task local state.
```

```
* ==============================================================%

new_type_abbrev ('program_counter', ":num");; new_type_abbrev
('registers', ":(num)list");;

new_type_abbrev ('task_local_state',
":program_counter#registers");;

new_type_abbrev ('local_state',
":(task_id#task_local_state)set");;

% --------------------------------------------------------------
    local state utility functions

-------------------------------------------------------------- %

% --------------------------------------------------------------
* get_ls_pc
-------------------------------------------------------------- %

let get_ls_pc = new_definition ('get_ls_pc', "
  (! (ls : local_state) (tid : task_id) .
    get_ls_pc ls tid = (FST (maps_to ls tid))
  )
");;

% --------------------------------------------------------------
* get_ls_regs
-------------------------------------------------------------- %

let get_ls_regs = new_definition ('get_ls_regs', "
  (! (ls : local_state) (tid : task_id) .
    get_ls_regs ls tid = (SND (maps_to ls tid))
  )
");;

% --------------------------------------------------------------
* put_ls_pc
-------------------------------------------------------------- %

let put_ls_pc = new_definition ('put_ls_pc', "
  (! (ls : local_state) (tid : task_id) (new_pc : program_counter) .
  put_ls_pc ls tid new_pc =
  let regs = (get_ls_regs ls tid)
  in
  let new_tls = (new_pc,regs)
  in
    (map_overwrite ls {(tid, new_tls)})
  )
");;


% ---------------------------------------------------------------------
* put_ls_regs
--------------------------------------------------------------------- %

let put_ls_regs = new_definition (`put_ls_regs`, "
  (! (ls : local_state) (tid : task_id) (regs : registers) .
  put_ls_regs ls tid regs =
  let pc = (get_ls_pc ls tid)
  in
  let new_tls = (pc,regs)
  in
    (map_overwrite ls {(tid, new_tls)})
  )
");;

% ---------------------------------------------------------------------
* put_ls_tls
--------------------------------------------------------------------- %

let put_ls_tls = new_definition (`put_ls_tls`, "
  (! (ls : local_state) (tid : task_id) (tls : task_local_state) .
  put_ls_tls ls tid tls =
    (map_overwrite ls {(tid, tls)})
  )
");;

% =====================================================================
*
*      global state types
*

The global state consists of the mailboxes and a sequencer used to
generate unique tags for messages.

Each mailbox is defined as a FIFO queue of messages. Messages that
```

```
are sent to the mailbox are placed by the system at the end of the
queue. Messages that are received from the mailbox by the
mailbox's owner are read from the current head of the list. A
''next'' pointer into the list keeps track of the current head.

Messages consist of the message data and the id of the sender. The
system automatically attaches the sender's id to the message data
as part of the SEND operation.

The complete set of mailboxes in the system are described as a
partial function from task_ids to mailboxes.

The global state is the complete set of mailboxes and the
sequencer.

* =============================================================%

new_type_abbrev ('sequencer', ":num");;

new_type_abbrev ('msg_data', ":num");; new_type_abbrev ('msg_id',
":sequencer");; new_type_abbrev ('msg_def',
":task_id#msg_data#msg_id");;

new_type_abbrev ('mbox_id', ":task_id");;

new_type_abbrev ('next', ":num");; new_type_abbrev ('mbox_def',
":next#((msg_def)list)");; new_type_abbrev ('mboxes',
":(mbox_id#mbox_def)set");; new_type_abbrev
('global_state',":mboxes#sequencer");;

% -------------------------------------------------------------
    global state utility functions

------------------------------------------------------------- %

% -------------------------------------------------------------
* cons_msg

Construct a message out of a sender id, message data, and message
id.
------------------------------------------------------------- %

let cons_msg = new_definition ('cons_msg', "
  (! (sndr : task_id) (mdata : msg_data) (msgid : msg_id) .
  cons_msg sndr mdata msgid =
    (sndr,mdata,msgid)
  )
");;

% ---------------------------------------------------------------
* get_msg_sndr

--------------------------------------------------------------- %

let get_msg_sndr = new_definition (`get_msg_sndr`, "
  (! (msg : msg_def).
    get_msg_sndr msg = (FST msg)
  )
");;

% ---------------------------------------------------------------
* get_msg_data

--------------------------------------------------------------- %

let get_msg_data = new_definition (`get_msg_data`, "
  (! (msg : msg_def).
    get_msg_data msg = (FST (SND msg))
  )
");;

% ---------------------------------------------------------------
* get_msg_id

--------------------------------------------------------------- %

let get_msg_id = new_definition (`get_msg_id`, "
  (! (msg : msg_def).
    get_msg_id msg = (SND (SND msg))
  )
");;

% ---------------------------------------------------------------
* cons_mbx

Construct a mailbox out of a list of messages and a next pointer.
--------------------------------------------------------------- %
```

```
let cons_mbx = new_definition (`cons_mbx`, "
  (! (nxt : *) (msgs : **) .
  cons_mbx nxt msgs =
    (nxt, msgs)
  )
");;

% -----------------------------------------------------------------
* get_mbx_next

----------------------------------------------------------------- %

let get_mbx_next = new_definition (`get_mbx_next`, "
  (! (mbx : mbox_def).
    get_mbx_next mbx = (FST mbx)
  )
");;

% -----------------------------------------------------------------
* get_mbx_msgs

----------------------------------------------------------------- %

let get_mbx_msgs = new_definition (`get_mbx_msgs`, "
  (! (mbx : mbox_def).
    get_mbx_msgs mbx = (SND mbx)
  )
");;


% -----------------------------------------------------------------
* get_gs_mbxs

----------------------------------------------------------------- %

let get_gs_mbxs = new_definition (`get_gs_mbxs`, "
  (! (gs : global_state).
    get_gs_mbxs gs = (FST gs)
  )
");;

% -----------------------------------------------------------------
* get_gs_seq
```

```
---------------------------------------------------------------- %

let get_gs_seq = new_definition (`get_gs_seq`, "
  (! (gs : global_state).
    get_gs_seq gs = (SND gs)
  )
");;

% ----------------------------------------------------------------
* put_gs_mbxs

---------------------------------------------------------------- %

let put_gs_mbxs = new_definition (`put_gs_mbxs`, "
  (! (gs : global_state) (new_mbxs : mboxes) .
  put_gs_mbxs gs new_mbxs =
  let seq = (get_gs_seq gs)
  in
    (new_mbxs, seq)
  )
");;

% ----------------------------------------------------------------
* get_mbxs_mbx

  Returns an empty mailbox if the specified mbox id is not in the set
  of mailboxes.
---------------------------------------------------------------- %

let get_mbxs_mbx = new_definition (`get_mbxs_mbx`, "
  (! (mbxs : mboxes) (mbxid : mbox_id) .
  get_mbxs_mbx mbxs mbxid =
    ((mbxid IN dom(mbxs)) => (maps_to mbxs mbxid) |
     (0,[]))
  )
");;

% ----------------------------------------------------------------
* get_gs_mbx

---------------------------------------------------------------- %

let get_gs_mbx = new_definition (`get_gs_mbx`, "
  (! (gs : global_state) (mbxid : mbox_id) .
  get_gs_mbx gs mbxid =
  let mbxs = (get_gs_mbxs gs)
  in
    (get_mbxs_mbx mbxs mbxid)
  )
");;

% ------------------------------------------------------------------
* put_mbx_msg

Takes a mbx and msg as inputs, returns the mbx with the msg
appended.
------------------------------------------------------------------ %

let put_mbx_msg = new_definition (`put_mbx_msg`, "
  (! (mbx:mbox_def)(msg:msg_def) .
  put_mbx_msg mbx msg =
  let msgs = (get_mbx_msgs mbx) and nxt = (get_mbx_next mbx)
  in
  let new_msgs = (SNOC msg msgs)
  in
    (cons_mbx nxt new_msgs)
  )
");;

% ------------------------------------------------------------------
* put_mbxs_mbx

Replaces a mailbox in a set of mboxes. Has no effect if the
specified mbox id is not in the domain of mboxes.
------------------------------------------------------------------ %

let put_mbxs_mbx = new_definition (`put_mbxs_mbx`, "
  (! (mbxs : mboxes) (mbxid : mbox_id) (new_mbx : mbox_def) .
  put_mbxs_mbx mbxs mbxid new_mbx =
    (map_replace mbxs {(mbxid, new_mbx)})
  )
");;

% ------------------------------------------------------------------
* add_mbxs_mbx

Adds a mailbox to a set of mailboxes. Overwrites an existing
mailbox with the same id.
```

```
-------------------------------------------------------------- %

let add_mbxs_mbx = new_definition (`add_mbxs_mbx`, "
  (! (mbxs : mboxes) (mbxid : mbox_id) (new_mbx : mbox_def) .
  add_mbxs_mbx mbxs mbxid new_mbx =
    (map_overwrite mbxs {(mbxid, new_mbx)})
  )
");;

% --------------------------------------------------------------
* put_mbxs_msg

Takes a destination mbx id, a set of mbxs and a msg as inputs.
Returns the set of mbxs with the msg appended to the destination
mbx.

Returns the input mbxs unchanged if the input mbox id is not in
the domain of mbxs.

-------------------------------------------------------------- %

let put_mbxs_msg = new_definition (`put_mbxs_msg`, "
  (! (mbxs:mboxes)(dst:mbox_id)(msg:msg_def) .
  put_mbxs_msg mbxs dst msg =
   let mbx = (maps_to mbxs dst)
   in
   let new_mbx = (put_mbx_msg mbx msg)
   in
     (put_mbxs_mbx mbxs dst new_mbx)
  )
");;


% --------------------------------------------------------------
* put_mbxs_msg_data

put_mbx_msg_data takes a mbox_id that represents the destination
mailbox of a message, message data, a message id, a task_id that
represents the sender of a message (viz., id of a system call) and
a set of mailboxes as inputs.

Its output is a new set of mailboxes where the message data, id
and sender's id have been delivered (in the form of a complete
message) to the destination mailbox.
```

```
Output set of mailboxes same as input if the destination is not
present in the set of mbxs.

----------------------------------------------------------------- %

let put_mbxs_msg_data = new_definition (`put_mbxs_msg_data`, "
  (! (dst:mbox_id)(mdata:msg_data)(mid:msg_id)(sndr:task_id)(mbxs :
mboxes).
  put_mbxs_msg_data dst mdata mid sndr mbxs =
  let new_msg = (cons_msg sndr mdata mid)
  in
    (put_mbxs_msg mbxs dst new_msg)
  )
");;



% -----------------------------------------------------------------
* put_gs_mbx

----------------------------------------------------------------- %

let put_gs_mbx = new_definition (`put_gs_mbx`, "
  (! (gs : global_state) (mbxid : mbox_id) (new_mbx : mbox_def) .
  put_gs_mbx gs mbxid new_mbx =
  let seq = (get_gs_seq gs) and mbxs = (get_gs_mbxs gs)
  in
  let new_mbxs = (put_mbxs_mbx mbxs mbxid new_mbx)
  in
    (new_mbxs,seq)
  )
");;

% -----------------------------------------------------------------
* put_gs_msg

Taking an input global state, put the input message into the
mailbox identified by the input mbxid. Return the modified global
state.

----------------------------------------------------------------- %

let put_gs_msg = new_definition (`put_gs_msg`, "
  (! (gs : global_state) (mbxid : mbox_id) (new_msg : msg_def) .
  put_gs_msg gs mbxid new_msg =
  let mbxs = (get_gs_mbxs gs)
  in
  let new_mbxs = (put_mbxs_msg mbxs mbxid new_msg)
  in
    (put_gs_mbxs gs new_mbxs)
  )
");;

% -----------------------------------------------------------------
* get_mbx_msg

Returns the message in the input mailbox pointed to by that
mailbox's ``next'' pointer.

UNDEFINED if next >= LENGTH(mbox) (i.e., if there are no unread
msgs). Safe if used with ``mbx_is_unread_msg'', as follows:

(mbx_is_unread_msg mbx) => (get_mbx_msg mbx) | ...

----------------------------------------------------------------- %

let get_mbx_msg = new_definition (`get_mbx_msg`, "
  (! (mbx : mbox_def) .
  get_mbx_msg mbx =
  let msgs = (get_mbx_msgs mbx) and nxt = (get_mbx_next mbx)
  in
    (EL nxt msgs)
  )
");;

% -----------------------------------------------------------------
* mbx_is_unread_msg

Returns TRUE if there is at least one unread msg in the input
mailbox, FALSE if there are no unread messages.

----------------------------------------------------------------- %

let mbx_is_unread_msg = new_definition (`mbx_is_unread_msg`, "
  (! (mbx : mbox_def) .
  mbx_is_unread_msg mbx =
  let msgs = (get_mbx_msgs mbx) and nxt = (get_mbx_next mbx)
  in
    (nxt < (LENGTH msgs))
  )
");;

% -----------------------------------------------------------------
* read_mbx_msg

Returns the message in the input mailbox pointed to by the
``next'' pointer. Automatically increments the "next" pointer and
returns the updated mailbox along with the message.

UNDEFINED if next >= LENGTH(msgs) (i.e., if there are no unread
msgs). Safe if used with ``mbx_is_unread_msg'', as follows:

(mbx_is_unread_msg mbx) => (read_mbx_msg mbx) | ...

----------------------------------------------------------------- %

let read_mbx_msg = new_definition (`read_mbx_msg`, "
  (! mbx .
    read_mbx_msg mbx =
    let msgs = (get_mbx_msgs mbx) and nxt = (get_mbx_next mbx)
    in
    let msg = (EL nxt msgs) and new_nxt = (nxt+1)
    in
      (msg, (cons_mbx new_nxt msgs))
  )
");;

% -----------------------------------------------------------------
* get_gs_msg

Returns the message in the specified mailbox pointed to by that
mailbox's ``next'' pointer.

UNDEFINED if next >= LENGTH(msgs) (i.e., if there are no unread
msgs). Safe if used with ``gs_is_unread_msgs'', as follows:

(gs_is_unread_msgs gs mbxid) => (get_gs_msg gs mbxid) | ...

----------------------------------------------------------------- %

let get_gs_msg = new_definition (`get_gs_msg`, "
  (! (gs : global_state) (mbxid : mbox_id) .
  get_gs_msg gs mbxid =
  let mbx = (get_gs_mbx gs mbxid)
  in
    (get_mbx_msg mbx)
  )
");;

% ---------------------------------------------------------------
* gs_is_unread_msg

Returns TRUE if there is at least one unread msg in the specified
mailbox, FALSE if there are no unread messages.

--------------------------------------------------------------- %

let gs_is_unread_msg = new_definition (`gs_is_unread_msg`, "
  (! (gs : global_state) (mbxid : mbox_id) .
  gs_is_unread_msg gs mbxid =
  let mbx = (get_gs_mbx gs mbxid)
  in
    (mbx_is_unread_msg mbx)
  )
");;

% ---------------------------------------------------------------
* inc_mbx_next

Increments the ``next'' pointer in a mailbox.

--------------------------------------------------------------- %

let inc_mbx_next = new_definition (`inc_mbx_next`, "
  (! (mbx : mbox_def) .
  inc_mbx_next mbx =
  let msgs = (get_mbx_msgs mbx) and nxt = (get_mbx_next mbx)
  in
    ((nxt+1), msgs)
  )
");;

% ---------------------------------------------------------------
* inc_mbxs_mbx_next
```

```
Increments the ``next'' pointer in a specified mailbox.

No effect if the specified mailbox does not exist.
---------------------------------------------------------------- %

let inc_mbxs_mbx_next = new_definition (`inc_mbxs_mbx_next`, "
  (! (mbxs : mboxes) (mbxid : mbox_id) .
  inc_mbxs_mbx_next mbxs mbxid =
  let mbx = (get_mbxs_mbx mbxs mbxid)
  in
  let incremented_mbox = (inc_mbx_next mbx)
  in
    (put_mbxs_mbx mbxs mbxid incremented_mbox)
  )
");;

% ----------------------------------------------------------------
* inc_gs_mbx_next

Increments the ``next'' pointer in a specified mailbox.

---------------------------------------------------------------- %

let inc_gs_mbx_next = new_definition (`inc_gs_mbx_next`, "
  (! (gs: global_state) (mbxid : mbox_id) .
  inc_gs_mbx_next gs mbxid =
  let mbx = (get_gs_mbx gs mbxid)
  in
  let incremented_mbox = (inc_mbx_next mbx)
  in
    (put_gs_mbx gs mbxid incremented_mbox)
  )
");;

% ----------------------------------------------------------------
* inc_gs_seq

Increments the global state sequencer.

---------------------------------------------------------------- %

let inc_gs_seq = new_definition (`inc_gs_seq`, "
  (! (gs : global_state).
  inc_gs_seq gs =
  let mbxs = (get_gs_mbxs gs) and seq = (get_gs_seq gs)
  in
    (mbxs, (seq+1))
  )
");;


% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define the complete system state
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %
% ================================================================
*
*       complete system state
*

The complete system state consists of the local state and the
global state.

* ===============================================================%

new_type_abbrev (`system_state`,":local_state#global_state");;

% ---------------------------------------------------------------
    system state utility functions

--------------------------------------------------------------- %

% ---------------------------------------------------------------
* get_ss_ls

--------------------------------------------------------------- %

let get_ss_ls = new_definition (`get_ss_ls`, "
  (! (ss : system_state).
    get_ss_ls ss = (FST ss)
  )
");;

% ---------------------------------------------------------------
* get_ss_gs

--------------------------------------------------------------- %

let get_ss_gs = new_definition (`get_ss_gs`, "
  (! (ss : system_state).
    get_ss_gs ss = (SND ss)
  )
");;

% -----------------------------------------------------------------
* put_ss_ls

Replaces the local state in a system state with the input local
state.
----------------------------------------------------------------- %

let put_ss_ls = new_definition (`put_ss_ls`, "
  (! (ss : system_state) (new_ls : local_state) .
    put_ss_ls ss new_ls =
    let gs = (get_ss_gs ss)
    in
      (new_ls,gs)
  )
");;

% -----------------------------------------------------------------
* put_ss_gs

Replaces the global state in a system state with the input global
state.
----------------------------------------------------------------- %

let put_ss_gs = new_definition (`put_ss_gs`, "
  (! (ss : system_state) (new_gs : global_state) .
    put_ss_gs ss new_gs =
    let ls = (get_ss_ls ss)
    in
      (ls,new_gs)
  )
");;


% -----------------------------------------------------------------
* good_mbxs

The conditions the mboxes must always satisfy in order to be
self-consistent. These conditions are as follows:
```

```
1) The set of mailboxes must be a mapping (viz., there must be a
one to one mapping between mailbox ids and the lists of messages).

2) For each mbox, the pointer to the next message to read must be
less than or equal to the length of the list of messages. (If the
pointer equals the length then the maibox is empty.)

--------------------------------------------------------------- %

let good_mbxs = new_definition (`good_mbxs`, "
  (! (mbxs : mboxes) .
  good_mbxs mbxs =
% condition 1%
    (is_mapping mbxs) /\
% condition 2%
    (!mbx . (mbx IN (rng mbxs)) ==>
         ((get_mbx_next mbx) <= (LENGTH (get_mbx_msgs mbx)))
    )
  )
");;

% ---------------------------------------------------------------
* good_gs

The conditions the global state must always satisfy in order to be
self-consistent. These conditions are as follows:

1) The mboxes must be self-consistent.

2) All messages in all mailboxes must have msg ids that are less
than the current value of the sequencer.

--------------------------------------------------------------- %

let good_gs = new_definition (`good_gs`, "
  (! (gs : global_state) .
  good_gs gs =
  let mbxs = (get_gs_mbxs gs) and seq = (get_gs_seq gs)
  in
% condition 1%
    ((good_mbxs mbxs) /\
% condition 2%
    (!mbx . (mbx IN (rng mbxs)) ==>
         (!msg . (MEMBER msg (get_mbx_msgs mbx)) ==>
          ((get_msg_id msg) < (seq))
        )
    ))
  )
");;

% ------------------------------------------------------------------
* good_ls

The conditions the local state must always satisfy in order to be
self-consistent. These conditions are as follows:

1) The local state must be a one-to-one mapping between task ids
and task local states.

---------------------------------------------------------------- %

let good_ls = new_definition (`good_ls`, "
  (! (ls : local_state).
  good_ls ls =
% condition 1%
    (is_mapping ls)
  )
");;

% ------------------------------------------------------------------
* good_sys_state

The conditions the system state must always satisfy in order to be
self-consistent. These conditions are as follows:

1) The global state is self-consistent.

2) The local state is self-consistent.

---------------------------------------------------------------- %

let good_sys_state = new_definition (`good_sys_state`, "
  (! (ss : system_state).
  good_sys_state ss =
  let ls = (get_ss_ls ss) and gs = (get_ss_gs ss)
  in (
% condition 1%
    (good_gs gs) /\
% condition 2%
    (good_ls ls)
  ))
");;

close_theory();;
```

## A.6 F-FG.ml

```
%-------------------------------------
* File:    F-FG.ml
* Version: 0.0
* Date:    05/20/98

Replaced F_func and G_func with universally quantified functions.

*---------------------------------------%

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
The composition of the F and FG servers.

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

loadt `aux/init.ml`;; new_theory_safe `F-FG`;; load_library
`sets`;; load_parent `silobasic`;;

% ================================================================
F and FG ids.

The mailbox and task ids of the F and FG servers.
============================================================== %

new_constant(`F_ID`, ":num");; new_constant(`FG_ID`, ":num");;

% ================================================================
F internal state.

The internal state of the F server specification. This state is
modifiable only by the F server specification, and not by the
environment.
```

```
Consists of the F work buffer. The work buffer is a single length
message buffer. A boolean flag is used to indicate if the buffer
is empty (FALSE) or full.

================================================================= %

new_type_abbrev (`F_work_buf`, ":msg_def");; new_type_abbrev
(`F_work_buf_flag`, ":bool");;

new_type_abbrev
(`F_internal_state`,":F_work_buf#F_work_buf_flag");;

% -----------------------------------------------------------------
    F state utility functions

----------------------------------------------------------------- %

% -----------------------------------------------------------------
* Fs_get_buf

Extracts the F buffer from an F state. Automatically resets the
flag.

----------------------------------------------------------------- %

let Fs_get_buf = new_definition (`Fs_get_buf`, "
  (! (Fs : F_internal_state).
    Fs_get_buf Fs =
    let (buf, flag) = Fs
    in
      (buf, (buf, F))
  )
");;


% -----------------------------------------------------------------
* get_Fs_buf

Extracts the F buffer from an F state.

----------------------------------------------------------------- %

let get_Fs_buf = new_definition (`get_Fs_buf`, "
  (! (Fs : F_internal_state).
    get_Fs_buf Fs = (FST Fs)
  )
");;


% ---------------------------------------------------------------
* Fs_get_flag

Extracts the F work buffer flag from an F state.

--------------------------------------------------------------- %

let Fs_get_flag = new_definition (`Fs_get_flag`, "
  (! (Fs : F_internal_state).
    Fs_get_flag Fs = (SND Fs)
  )
");;

% ---------------------------------------------------------------
* Fs_put_buf

Replaces the F work buffer in a F state with the input msg.
Automatically sets flag to TRUE.
--------------------------------------------------------------- %

let Fs_put_buf = new_definition (`Fs_put_buf`, "
  (! (Fs : F_internal_state) (new_msg : msg_def) .
    Fs_put_buf Fs new_msg =
      (new_msg,T)
  )
");;

% ===============================================================
FG internal state.

The internal state of the FG server specification. This state is
modifiable only by the FG server specification, and not by the
environment.

The FG internal state consists of a work buffer for the current
message that has just been read, and a pending queue of messages
whose responses have not yet been sent. A boolean flag is used to
indicate if the buffer is full (TRUE) or not. A "next" pointer is
used to keep track of the head of the queue.
```

```
================================================================ %

new_type_abbrev ('FG_buf', ":msg_def");; new_type_abbrev
('FG_flag', ":bool");;

new_type_abbrev ('FG_queue', ":(msg_def)list");;

new_type_abbrev ('FG_internal_state',":FG_buf#FG_flag#FG_queue");;

% ----------------------------------------------------------------
    FG state utility functions

---------------------------------------------------------------- %

% ----------------------------------------------------------------
* FGs_get_buf

Extracts the FG buffer from an FG state. Automatically resets the
flag.

---------------------------------------------------------------- %

let FGs_get_buf = new_definition ('FGs_get_buf', "
  (! (FGs : FG_internal_state).
    FGs_get_buf FGs =
    let (buf, flag, que) = FGs
    in
      (buf, (buf, F, que))
  )
");;

% ----------------------------------------------------------------
* get_FGs_buf

Extracts the FG buffer from an FG state.

---------------------------------------------------------------- %

let get_FGs_buf = new_definition ('get_FGs_buf', "
  (! (FGs : FG_internal_state).
    get_FGs_buf FGs = (FST FGs)
  )
");;
```

```
% --------------------------------------------------------------
* FGs_get_flag

Extracts the FG flag from an FG state.

------------------------------------------------------------ %

let FGs_get_flag = new_definition (`FGs_get_flag`, "
  (! (FGs : FG_internal_state).
    FGs_get_flag FGs = (FST (SND FGs))
  )
");;

% --------------------------------------------------------------
* FGs_get_queue

Extracts the FG queue from an FG state.

------------------------------------------------------------ %

let FGs_get_queue = new_definition (`FGs_get_queue`, "
  (! (FGs : FG_internal_state).
    FGs_get_queue FGs = (SND (SND FGs))
  )
");;

% --------------------------------------------------------------
* FGs_put_buf

Replaces the FG buffer in a FG state with the input msg.
Automatically sets flag to TRUE.

------------------------------------------------------------ %

let FGs_put_buf = new_definition (`FGs_put_buf`, "
  (! (FGs : FG_internal_state) (msg : msg_def) .
    FGs_put_buf FGs msg =
    let (buf, flag, que) = FGs
    in
      (msg, T, que)
  )
");;
```

```
% ----------------------------------------------------------------
* FGs_put_queue_msg

Adds the input message to the end of the queue.

---------------------------------------------------------------- %

let FGs_put_queue_msg = new_definition (`FGs_put_queue_msg`, "
  (! (FGs : FG_internal_state) (msg : msg_def) .
    FGs_put_queue_msg FGs msg =
    let (buf, flag, que) = FGs
    in
      (buf, flag, (SNOC msg que))
  )
");;

% ----------------------------------------------------------------
* FGs_is_pending

Returns TRUE if there is at least one unread msg in the FG queue
(i.e., for which a response is pending).

---------------------------------------------------------------- %

let FGs_is_pending = new_definition (`FGs_is_pending`, "
  (! (FGs : FG_internal_state) .
    FGs_is_pending FGs =
    let (buf, flag, que) = FGs
    in
      (0 < LENGTH que)
  )
");;


% ----------------------------------------------------------------
* FGs_get_queue_msg

Returns the message at the head of the queue. Automatically
removes the message from the queue and returns the updated FG
state along with the msg.

UNDEFINED if there are no unread msgs. Safe if used with
``FGs_is_pending'', as follows:
```

```
(FGs_is_pending FGs) => (FGs_get_queue_msg FGs) | ...

-------------------------------------------------------------- %

let FGs_get_queue_msg = new_definition ('FGs_get_queue_msg', "
  (! (FGs : FG_internal_state) .
    FGs_get_queue_msg FGs =
    let (buf, flag, que) = FGs
    in
    let msg = HD que
    and new_mbx = TL que
    in
      (msg, (buf, flag, new_mbx))
  )
");;



% ===============================================================
*
*       Trace definition
*

The specification for a system is a set of all possible traces of
events in that system. Traces are represented as a list of
(agents, state) pairs, called "trace_elements". The agent in a
trace_element performed an atomic action to move the system from
the preceding state to the state in the trace_element.

Agents can be either part of the environment or the system that we
are specifying.

The state in the first trace_element in a trace is the initial
system state. The agent in the first trace_element is undefined
(because there is no preceding state from which to transition to
the initial state).

The system state for this example consists of the global state in
the system, plus the system's internal state.
* ==============================================================%

let Agent_Axiom =
    save_thm('Agent_Axiom',
           define_type 'Agent'
```

```
                 `Agent = ENV
                        | SYS_F
                        | SYS_FG
`);;

let Agent_INDUCT =
    save_thm(`Agent_INDUCT`,
        prove_induction_thm Agent_Axiom);;
let Agent_CASES =
    save_thm(`Agent_CASES`,
        prove_cases_thm Agent_INDUCT);;
let Agent_DISTINCT =
    save_thm(`Agent_DISTINCT`,
        prove_constructors_distinct Agent_Axiom);;

let TLabel_Axiom =
    save_thm(`TLabel_Axiom`,
             define_type `TLabel`
                `TLabel = FREADS
                        | FRESPONDS
                        | FGREADS
                        | FGRESPONDS
                        | STUTTER
`);;

let TLabel_INDUCT =
    save_thm(`TLabel_INDUCT`,
        prove_induction_thm TLabel_Axiom);;
let TLabel_CASES =
    save_thm(`TLabel_CASES`,
        prove_cases_thm TLabel_INDUCT);;
let TLabel_DISTINCT =
    save_thm(`TLabel_DISTINCT`,
        prove_constructors_distinct TLabel_Axiom);;


new_type_abbrev (`trace_istate`,
":F_internal_state#FG_internal_state");; new_type_abbrev
(`trace_state`, ":mboxes#trace_istate");;

new_type_abbrev (`trace_element`, ":Agent#TLabel#trace_state");;
new_type_abbrev (`trace_def`, ":num->trace_element");;

% ---------------------------------------------------------------
```

```
    trace utility functions

These first functions are not directly used by the specifications.

---------------------------------------------------------------- %

% ----------------------------------------------------------------
* get_trace_istate

Extracts the internal state from the trace state.

---------------------------------------------------------------- %

let get_trace_istate = new_definition (`get_trace_istate`, "
  (! (ts : trace_state) .
    get_trace_istate ts = (SND ts)
  )
");;

% ----------------------------------------------------------------
* put_trace_istate

---------------------------------------------------------------- %

let put_trace_istate = new_definition (`put_trace_istate`, "
  (! (ts : trace_state) (new_is : trace_istate) .
    put_trace_istate ts new_is =
    let (mbxs, old_is) = ts
    in
      (mbxs, new_is)
  )
");;

% ----------------------------------------------------------------
* get_is_Fs

---------------------------------------------------------------- %

let get_is_Fs = new_definition (`get_is_Fs`, "
  (! (is : trace_istate) .
    get_is_Fs is =
    let (Fs, FGs) = is
    in
      (Fs)
  )
");;

% -----------------------------------------------------------------
* get_is_FGs

----------------------------------------------------------------- %

let get_is_FGs = new_definition (`get_is_FGs`, "
  (! (is : trace_istate) .
    get_is_FGs is =
    let (Fs, FGs) = is
    in
      (FGs)
  )
");;

% -----------------------------------------------------------------
* put_is_Fs

----------------------------------------------------------------- %

let put_is_Fs = new_definition (`put_is_Fs`, "
  (! (is : trace_istate) (new_Fs : F_internal_state) .
    put_is_Fs is new_Fs =
    let (old_Fs, FGs) = is
    in
      (new_Fs, FGs)
  )
");;

% -----------------------------------------------------------------
* put_is_FGs

----------------------------------------------------------------- %

let put_is_FGs = new_definition (`put_is_FGs`, "
  (! (is : trace_istate) (new_FGs : FG_internal_state) .
    put_is_FGs is new_FGs =
    let (Fs, old_FGs) = is
    in
      (Fs, new_FGs)
  )
");;
```

```
% -----------------------------------------------------------------
    trace utility functions used by specifications

----------------------------------------------------------------- %

% -----------------------------------------------------------------
* get_trace_agent

----------------------------------------------------------------- %

let get_trace_agent = new_definition (`get_trace_agent`, "
  (! (tel : trace_element).
    get_trace_agent tel = (FST tel)
  )
");;

% -----------------------------------------------------------------
* get_trace_tlabel

----------------------------------------------------------------- %

let get_trace_tlabel = new_definition (`get_trace_tlabel`, "
  (! (tel : trace_element).
    get_trace_tlabel tel = FST(SND tel)
  )
");;

% -----------------------------------------------------------------
* get_trace_state

----------------------------------------------------------------- %

let get_trace_state = new_definition (`get_trace_state`, "
  (! (tel : trace_element).
    get_trace_state tel = SND(SND tel)
  )
");;

% -----------------------------------------------------------------
* get_trace_mbxs

----------------------------------------------------------------- %
```

```
let get_trace_mbxs = new_definition (`get_trace_mbxs`, "
  (! (ts : trace_state) .
    get_trace_mbxs ts = (FST ts)
  )
");;

% ---------------------------------------------------------------
* get_trace_not_mbxs

--------------------------------------------------------------- %

let get_trace_not_mbxs = new_definition (`get_trace_not_mbxs`, "
  (! (ts : trace_state) .
    get_trace_not_mbxs ts = (SND ts)
  )
");;

% ---------------------------------------------------------------
* get_trace_Fs

--------------------------------------------------------------- %

let get_trace_Fs = new_definition (`get_trace_Fs`, "
  (! (ts : trace_state) .
    get_trace_Fs ts = (get_is_Fs (get_trace_istate ts))
  )
");;

% ---------------------------------------------------------------
* get_trace_FGs

--------------------------------------------------------------- %

let get_trace_FGs = new_definition (`get_trace_FGs`, "
  (! (ts : trace_state) .
    get_trace_FGs ts = (get_is_FGs (get_trace_istate ts))
  )
");;

% ---------------------------------------------------------------
* put_trace_mbxs

--------------------------------------------------------------- %
```

```
let put_trace_mbxs = new_definition (`put_trace_mbxs`, "
  (! (ts : trace_state) (new_mbxs : mboxes) .
    put_trace_mbxs ts new_mbxs =
    let (old_mbxs, is) = ts
    in
      (new_mbxs, is)
  )
");;

% ------------------------------------------------------------
* put_trace_Fs

------------------------------------------------------------ %

let put_trace_Fs = new_definition (`put_trace_Fs`, "
  (! (ts : trace_state) (new_Fs : F_internal_state) .
    put_trace_Fs ts new_Fs =
    let (mbxs, old_is) = ts
    in
    let (old_Fs, FGs) = old_is
    in
    let new_is = (new_Fs, FGs)
    in
      (mbxs, new_is)
  )
");;

% ------------------------------------------------------------
* put_trace_FGs

------------------------------------------------------------ %

let put_trace_FGs = new_definition (`put_trace_FGs`, "
  (! (ts : trace_state) (new_FGs : FG_internal_state) .
    put_trace_FGs ts new_FGs =
    let (mbxs, old_is) = ts
    in
    let (Fs, old_FGs) = old_is
    in
    let new_is = (Fs, new_FGs)
    in
      (mbxs, new_is)
  )
");;
```

```
% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define FG environment state transition relation
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% ================================================================
*
*       environment state transition relation (safety properties)
*

These functions and predicates define the state transition
relation (safety properties) for the environment. In this case,
the environment consists of the OS and other tasks.

* ================================================================%

% ----------------------------------------------------------------
* FG_env_invariant

The conditions of the system state, once established for the
initial state, that must be preserved by all state transitions as
an invariant

---------------------------------------------------------------- %

let FG_env_invariant = new_definition (`FG_env_invariant`, "
  (! (mbxs : mboxes) .
  FG_env_invariant mbxs =
% F server must have mailbox %
      (F_ID IN (dom mbxs)) /\
% FG server must have mailbox %
      (FG_ID IN (dom mbxs))
  )
");;

% ----------------------------------------------------------------
* FG_env_SEND

Returns TRUE if the cause of the state transition between ss1 and
ss2 is a message arriving in the FG mailbox. The message must have
a sender id other than FG_ID, because the FG server did not send
the message.
```

```
--------------------------------------------------------------- %

let FG_env_SEND = new_definition (`FG_env_SEND`, "
  (! (ss1 ss2 : trace_state) .
  FG_env_SEND ss1 ss2 =
    let (mbxs1, FGs1) = ((get_trace_mbxs ss1), (get_trace_FGs ss1))
    and (mbxs2, FGs2) = ((get_trace_mbxs ss2), (get_trace_FGs ss2))
    in
    let FGmbx1 = (get_mbxs_mbx mbxs1 FG_ID)
    and FGmbx2 = (get_mbxs_mbx mbxs2 FG_ID)
    in (
      ((dom mbxs1) = (dom mbxs2))  /\
      (FGs1 = FGs2) /\
      (? msg .
        let sndr = (get_msg_sndr msg)
        in (
          (sndr IN (dom mbxs1)) /\
          ~(sndr = FG_ID) /\
          ((put_mbx_msg FGmbx1 msg) = FGmbx2)
        )
      )
    )
  )
");;

% ---------------------------------------------------------------
* FG_env_arb

Returns TRUE if the cause of the state transition between ss1 and
ss2 is a state change to any state components other than the FG
mailbox, the FG internal state, and the domain of the mailboxes.

--------------------------------------------------------------- %

let FG_env_arb = new_definition (`FG_env_arb`, "
  (! (ss1 ss2 : trace_state) .
  FG_env_arb ss1 ss2 =
    let (mbxs1, FGs1) = ((get_trace_mbxs ss1), (get_trace_FGs ss1))
    and (mbxs2, FGs2) = ((get_trace_mbxs ss2), (get_trace_FGs ss2))
    in
    let FGmbx1 = (get_mbxs_mbx mbxs1 FG_ID)
    and FGmbx2 = (get_mbxs_mbx mbxs2 FG_ID)
    in (
      ((dom mbxs1) = (dom mbxs2))  /\
      (FGs1 = FGs2) /\
      (FGmbx1 = FGmbx2)
    )
  )
");;


% -----------------------------------------------------------------
* FG_env_safety

In english:

For all elements of the trace, t1 and t2, where t2 is the
immediate successor to t1 and where the agent in t2 is not SYS_FG
  the states in t1 and t2 must be identical
  or the state transition must satisfy the safety
  properties for one of the possible state transitions

Valid state transitions for the FG environment are as follows:

1) A message arriving in the FG mailbox, but the sender id may not
be FG_ID.

2) Any other state change so long as the FG mailbox, FG private
state, and domain of mailboxes are not affected.

No environment state transition may affect the internal state of
the FG server.

----------------------------------------------------------------- %

let FG_env_safety = new_definition (`FG_env_safety`,
  "(! (trace : trace_def).
  FG_env_safety trace =
    (! (i : num) .
      let t1 = (trace i)
      and t2 = (trace(SUC i))
      in (
        ~((get_trace_agent t2) = SYS_FG) ==>
        let ss1 = (get_trace_state t1)
        and ss2 = (get_trace_state t2)
        in (
% no system state change or %
          (ss1 = ss2) \/
```

```
% a message arrives in the FG mailbox %
          (FG_env_SEND ss1 ss2) \/
% any other state change that does not affect FG
  but still satisfies the invariant %
          (FG_env_arb ss1 ss2)
        )
      )
    )
  )"
);;



% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define FG server state transition relation
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% ===============================================================
*
*       FG server state transition relation (safety properties)
*

These functions and predicates define the atomic state transition
relation (safety properties) for the FG server.

These transitions consist of the FG server reading a message from
its mailbox and sending a response message.

* ==============================================================%


% --------------------------------------------------------------
* FG_reads_msg

FG reads a message from its mailbox and puts it in the work
buffer.

FG_reads_msg takes a global state (mailboxes) and FG internal
state as input.

Its output is a new system state where the next readable message
in the FG system mailbox has been read and placed into the "work"
buffer and the "next" pointer of the FG mailbox has been advanced.
If there is no unread message in the mailbox, or if there is
```

```
already a message in the work buffer, the input system state is
returned unchanged.

------------------------------------------------------------------ %

let FG_reads_msg = new_definition (`FG_reads_msg`, "
  (! ss .
  FG_reads_msg ss =
  let (mbxs, FGs) = ((get_trace_mbxs ss), (get_trace_FGs ss))
  in
  let mbx = (get_mbxs_mbx mbxs FG_ID)
  in
% Check if no messages or if message already in work buffer.
  If so, return state unchanged.                        %
    (((~mbx_is_unread_msg mbx) \/ (FGs_get_flag FGs)) => ss |
% Otherwise, read message... %
    (let (msg, new_mbx) = (read_mbx_msg mbx)
    in
% put back the modified mailbox %
    let new_mbxs = (put_mbxs_mbx mbxs FG_ID new_mbx)
% put msg in work buffer %
    and new_FGs = (FGs_put_buf FGs msg)
    in
      (put_trace_mbxs (put_trace_FGs ss new_FGs) new_mbxs)
    ))
  )
");;

% ------------------------------------------------------------------
* FG_reads

True if the second system state is reached from the first due to
FG reading a message from its mailbox.

------------------------------------------------------------------ %

let FG_reads = new_definition (`FG_reads`, "
  (! ss1 ss2 .
  FG_reads ss1 ss2 =
    (ss2 = (FG_reads_msg ss1))
  )
");;
```

```
% ------------------------------------------------------------------
* FG_push_request

FG pushes a request message onto its queue and sends a request to
the F server.

------------------------------------------------------------------ %

let FG_push_request = new_definition ('FG_push_request', "
  (! ss (G_func:num->num).
  FG_push_request ss G_func =
    let (mbxs, FGs) = ((get_trace_mbxs ss), (get_trace_FGs ss))
    in
    let (bufmsg, FGs1) = (FGs_get_buf FGs) % FGs1 has cleared buffer%
    in
% put request message on queue %
    let new_FGs = (FGs_put_queue_msg FGs1 bufmsg)
    in
% create message to send to F %
    let bufmdata = (get_msg_data bufmsg) and bufmid =
    (get_msg_id bufmsg)
    in
    let msg_for_F = (cons_msg FG_ID (G_func bufmdata) bufmid)
    in
% send message to F %
    let new_mbxs = (put_mbxs_msg mbxs F_ID msg_for_F)
    in
      (put_trace_mbxs (put_trace_FGs ss new_FGs) new_mbxs)
  )
");;



% ------------------------------------------------------------------
* FG_send_response

FG sends a message in response to a request message.

The request message should be at the head of the FG queue. Data
for the response message comes out of the message in the buffer.
The destination for the response message is the sender of the
original request.
```

```
Output is a new system state where the request message has been
removed from the queue and the response messages has been placed
in the destination mailbox.

If there is no message in the queue, the buffer is emptied and no
response message is sent. This condition should never happen if
the F server works properly, but is included in the spec to insure
that the resulting state is always defined.

------------------------------------------------------------- %

let FG_send_response = new_definition (`FG_send_response`,
  "(! ss .
  FG_send_response ss =
    let (mbxs, FGs) = ((get_trace_mbxs ss), (get_trace_FGs ss))
    in
    let (bufmsg, FGs1) = (FGs_get_buf FGs)
    in
% FGs1 has cleared buffer %
      ((FGs_is_pending FGs1) =>
        (let (rqst, new_FGs) = (FGs_get_queue_msg FGs1)
         in
         let bufmdata = (get_msg_data bufmsg)
         in
         let rsrc = (get_msg_sndr rqst) and rmid = (get_msg_id rqst)
         in
% create response message %
         let response = (cons_msg FG_ID bufmdata rmid)
         in
% send response message %
         let new_mbxs = (put_mbxs_msg mbxs rsrc response)
         in
           (put_trace_mbxs (put_trace_FGs ss new_FGs) new_mbxs)) |
        (put_trace_FGs ss FGs1)
      )
  )"
);;

% -------------------------------------------------------------
* FG_responds_msg

FG responds to the message in its "work" buffer.

FG_responds_msg takes a system state as input.
```

If the message in the buffer is from a task other than the F server, FG pushes the message on its queue and sends a message to the F server that contains G(v) as its data, where ''v'' was the data value in the original message.

If the message in the buffer is from the F server, then it is a response from the F server to an earlier message sent by FG, and contains a data value F(G(v)). The original request message that started this whole process (and that contains v) should be at the head of the FG queue. FG generates a response message for this request, containing the F(G(v)) data value and the original id of the request message, and removes the request from the queue.

The output is a new system state where the buffer has been emptied, the rest of the FG internal state was modified as described above, and a request message sent to the F server or a response message sent to another task. If there is no message already in the work buffer, the input system state is returned unchanged.

```
------------------------------------------------------------ %

let FG_responds_msg = new_definition (`FG_responds_msg`, "
  (! ss (G_func:num->num).
  FG_responds_msg ss G_func =
  let (mbxs, FGs) = ((get_trace_mbxs ss), (get_trace_FGs ss))
  in
% Check if work buffer has a message.%
%  If not, return state unchanged.   %
    ((~(FGs_get_flag FGs)) => ss |
% Otherwise, check who sent the message %
    (let (bufmsg, FGs1) = (FGs_get_buf FGs) % FGs1 has cleared buffer%
    in
    let bufsrc = (get_msg_sndr bufmsg)
    in
% If sender is not F, %
% put request message on queue and send message to F %
      ((~(bufsrc = F_ID)) => (FG_push_request ss G_func) |
% Otherwise, F sent the message. %
% remove rqst message from queue and send response. %
      (FG_send_response ss)
    )))
  )
```

```
");;

% ---------------------------------------------------------------
* FG_responds

True if the second system state is reached from the first due to
FG responding to a message in its work buffer.

--------------------------------------------------------------- %

let FG_responds = new_definition (`FG_responds`, "
  (! ss1 ss2 (G_func:num->num).
  FG_responds ss1 ss2 G_func =
    (ss2 = (FG_responds_msg ss1 G_func))
  )
");;


% ---------------------------------------------------------------
* FG_safety

In english:

For all elements of the trace, t1 and t2, where t2 is the
immediate successor to t1 and where the agent in t2 is FG
  the states in t1 and t2 must be identical
  or the state transition must satisfy the safety
  properties for one of the possible state transitions

Valid state transitions for FG are as follows:

Reading a message from the FG system mailbox and sending a
response message to the sender.

--------------------------------------------------------------- %

let FG_safety = new_definition (`FG_safety`,
  "(! (trace : trace_def) (G_func:num->num).
  FG_safety trace G_func =
    (! (i : num) .
      let t1 = (trace i)
      and t2 = (trace(SUC i))
      in (
        ((get_trace_agent t2) = SYS_FG) ==>
        let ss1 = (get_trace_state t1)
        and ss2 = (get_trace_state t2)
        in (
% no state change %
          (((get_trace_tlabel t2) = STUTTER) /\ (ss1 = ss2)) \/
% or a legal transition %
          (((get_trace_tlabel t2) = FGREADS) /\ (FG_reads ss1 ss2)) \/
          (((get_trace_tlabel t2) = FGRESPONDS) /\
           (FG_responds ss1 ss2 G_func))
        )
      )
    )
  )"
);;

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define FG server progress properties
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% ==============================================================
*
*       FG server state progress properties
*

These properties assert that the FG server will eventually read
all messages and send responses.

Assuming that the work buffer is eventually emptied, performing a
FG_read operation will read the next unread message in FG's
mailbox. FG_reads_progress guarantees that the read will
eventually happen. An (eventually) empty work buffer is guaranteed
by the progress property on the FG_responds operation. In this way
we guarantee that all input messages (requests) will eventually be
read and generate responses.

The progress properties for the FG server alone do not specify
that a response message will ever be generated. That depends on
the F server sending a response the FG server.

This method of specifying progress properties is possible because
of the way we wrote the safety properties. They were written to
only cause a state change when conditions were correct so that the
operation could be performed. Otherwise, performing either
operation causes no state change. Thus, our progress properties
```

```
specify "busy-waiting" on the part of the server.

This method is essentially that of UNITY.

* ===============================================================%

% ---------------------------------------------------------------
* FG_reads_progress

A progress property for the ``FG'' system call that asserts that,
starting from any point in the trace, FG will eventually perform
an FG_reads operation.

In english:

For all ``i'' (viz., a trace_element in a trace)
  there is some ``j'' (also a trace_element in the trace)
    such that j is later in the trace than i
    and whose agent is SYS_FG (i.e., the agent responsible for the
                               transition)
    and the reason for the state transition is
      a FG_reads_msg step
    and in all intermediate steps between i and j
      a FG_reads_msg step did not occur.

---------------------------------------------------------------- %

let FG_reads_progress = new_definition (`FG_reads_progress`,
  "(! (trace : trace_def) .
    FG_reads_progress trace =
      (! (i : num) .
        (? (j : num) .
          (i <= j) /\
      (get_trace_state(trace (SUC j)) =
       (FG_reads_msg (get_trace_state(trace j))))
        )
      )
  )"
);;


% ---------------------------------------------------------------
* FG_responds_progress
```

```
A progress property for the ``FG'' system call that asserts that,
starting from any point in the trace, FG will eventually perform
an FG_responds operation.

In english:

For all ``i'' (viz., a trace_element in a trace)
  there is some ``j'' (also a trace_element in the trace)
    such that j is later in the trace than i
    and whose agent is SYS_FG (i.e., the agent responsible for the
                                    transition)
    and the reason for the state transition is
      a FG_responds_msg step
    and in all intermediate steps between i and j
      a FG_responds_msg step did not occur.

----------------------------------------------------------------- %

let FG_responds_progress = new_definition (`FG_responds_progress`,
  "(! (trace : trace_def) (G_func:num->num) .
    FG_responds_progress trace G_func =
      (! (i : num) .
        (? (j : num) .
          (i <= j) /\
          (get_trace_state(trace (SUC j)) =
           (FG_responds_msg (get_trace_state(trace j)) G_func))
        )
      )
  )"
);;


% -----------------------------------------------------------------
* FG_progress

The progress properties for the FG system call.

----------------------------------------------------------------- %
%
let FG_progress = new_definition (`FG_progress`, "
  (! (trace : trace_def) .
  FG_progress trace =
    (FG_reads_progress trace) /\
    (FG_responds_progress trace)
  )
");;
%
% ----------------------------------------------------------------
* FG_env_progress

The progress properties for the FG system call environment.

There are no progress properties defined because the server is
purely reactive, so this property is always TRUE.

---------------------------------------------------------------- %

let FG_env_progress = new_definition (`FG_env_progress`, "
  (! (trace : trace_def) .
  FG_env_progress trace =
    T
  )
");;

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define FG initial state
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% ================================================================
*
*       initial state
*

*       environmental requirements for initial system state

The requirements on the system state consist of structural
consitency requirements on the data, which must remain invariant
in every state in a trace, and initial policy conditions (e.g.,
that mailboxes are initially empty).

The environment does not specify any initial conditions on the FG
server's internal state.

*       FG's requirements for initial internal state

FG's requirements for its initial internal state are that its work
buffer is empty.
```

```
* ==============================================================%

% --------------------------------------------------------------
* FG_good_init_mbxs

The environmental requirements on the intial system state that are
due to system policy.

We require that all mailboxes are initially empty. We require this
as a convenience, since it is easier than, for example, specifying
the relationship between request and response messages in the
initial state.

-------------------------------------------------------------- %

let FG_good_init_mbxs = new_definition (`FG_good_init_mbxs`, "
  (! (mbxs : mboxes) .
  FG_good_init_mbxs mbxs =
    (! mbx . (mbx IN (rng mbxs)) ==>
     (~mbx_is_unread_msg mbx)
    )
  )
");;

% --------------------------------------------------------------
* FG_env_init

The complete definition of a good initial system state.

The initial system state is correct if it is self-conistent and if
it satisfies all policy requirements.

The consistency of the system state, once established for the
initial state, must be preserved by all state transitions as an
invariant (i.e., we have to prove that it is invariant).

-------------------------------------------------------------- %

let FG_env_init = new_definition (`FG_env_init`, "
  (! (trace : trace_def) .
  FG_env_init trace =
% need unique mailboxes for the servers %
    ~(F_ID = FG_ID) /\
% require at least one trace element so we have an initial state %
%    ((LENGTH trace) > 0) /\
    let init_t = (EL 0 trace)%
    let init_t = (trace 0)
    in
    let mbxs = (get_trace_mbxs (get_trace_state init_t))
    in (
      (FG_env_invariant mbxs) /\
% self-consistency %
      (good_mbxs mbxs) /\
% policy req's %
      (FG_good_init_mbxs mbxs)
    )
  )
");;


% ---------------------------------------------------------------
* FG_init

The definition of a good internal state for the FG server.

The buffer and queue must be empty (i.e., there are no responses
pending).

FG cannot specify any initial conditions on the mailboxes, whose
initial condition is reserved for the environment.

--------------------------------------------------------------- %

let FG_init = new_definition (`FG_init`, "
  (! (trace : trace_def) .
  FG_init trace =
% no need to require at least one trace element because already
specified by FG_env_init. %
%    let init_t = (EL 0 trace)%
    let init_t = (trace 0)
    in
    let FGs = (get_trace_FGs (get_trace_state init_t))
    in (
      (~FGs_get_flag FGs) /\
      (~FGs_is_pending FGs)
  ))
");;
```

```
% ------------------------------------------------------------------
* FG_system_specification

------------------------------------------------------------------ %
%
let FG_system_specification = new_definition
('FG_system_specification', "
  FG_system_specification =
    { trace : trace_def |
      ((FG_env_init trace) /\
      (FG_env_safety trace) /\
      (FG_env_progress trace)) ==>
          ((FG_init trace) /\
          (FG_safety trace) /\
          (FG_progress trace))
    }
");;
%

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define F environment state transition relation
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% ==================================================================
*
*       environment state transition relation (safety properties)
*

These functions and predicates define the state transition
relation (safety properties) for the environment. In this case,
the environment consists of the OS and other tasks.

* =================================================================%

% ------------------------------------------------------------------
* F_env_invariant

The conditions of the system state, once established for the
initial state, that must be preserved by all state transitions as
an invariant

------------------------------------------------------------------ %
```

```
let F_env_invariant = new_definition (`F_env_invariant`, "
  (! (mbxs : mboxes) .
  F_env_invariant mbxs =
% F server must have mailbox %
      (F_ID IN (dom mbxs))
  )
");;

% ----------------------------------------------------------------
* F_env_SEND

Returns TRUE if the cause of the state transition between ss1 and
ss2 is a message arriving in the F mailbox. The message must have
a sender id other than F_ID, because the F server did not send the
message.

---------------------------------------------------------------- %

let F_env_SEND = new_definition (`F_env_SEND`, "
  (! (ss1 ss2 : trace_state) .
  F_env_SEND ss1 ss2 =
    let (mbxs1, Fs1) = ((get_trace_mbxs ss1), (get_trace_Fs ss1))
    and (mbxs2, Fs2) = ((get_trace_mbxs ss2), (get_trace_Fs ss2))
    in
    let Fmbx1 = (get_mbxs_mbx mbxs1 F_ID)
    and Fmbx2 = (get_mbxs_mbx mbxs2 F_ID)
    in (
      ((dom mbxs1) = (dom mbxs2))  /\
      (Fs1 = Fs2) /\
      (? msg .
        let sndr = (get_msg_sndr msg)
        in (
          (sndr IN (dom mbxs1)) /\
          ~(sndr = F_ID) /\
          ((put_mbx_msg Fmbx1 msg) = Fmbx2)
        )
      )
    )
  )
");;

% ----------------------------------------------------------------
* F_env_arb
```

```
Returns TRUE if the cause of the state transition between ss1 and
ss2 is a state change to any state components other than the F
mailbox, the F internal state, and the domain of the mailboxes.

---------------------------------------------------------------- %

let F_env_arb = new_definition ('F_env_arb', "
  (! (ss1 ss2 : trace_state) .
  F_env_arb ss1 ss2 =
    let (mbxs1, Fs1) = ((get_trace_mbxs ss1), (get_trace_Fs ss1))
    and (mbxs2, Fs2) = ((get_trace_mbxs ss2), (get_trace_Fs ss2))
    in
    let Fmbx1 = (get_mbxs_mbx mbxs1 F_ID)
    and Fmbx2 = (get_mbxs_mbx mbxs2 F_ID)
    in (
      ((dom mbxs1) = (dom mbxs2))  /\
      (Fs1 = Fs2) /\
      (Fmbx1 = Fmbx2)
    )
  )
");;


% -----------------------------------------------------------------
* F_env_safety

In english:

For all elements of the trace, t1 and t2, where t2 is the
immediate successor to t1 and where the agent in t2 is not SYS_F
  the states in t1 and t2 must be identical
  or the state transition must satisfy the safety
  properties for one of the possible state transitions

Valid state transitions for the F environment are as follows:

1) A message arriving in the F mailbox, but the sender id may not
be F_ID.

2) Any other state change so long as the F mailbox, F private
state, and domain of mailboxes are not affected.

No environment state transition may affect the internal state of
```

```
the F server.

------------------------------------------------------------------ %

let F_env_safety = new_definition (`F_env_safety`, "
  (! (trace : trace_def).
  F_env_safety trace =
    (! (i : num) .
      let t1 = (trace i)
      and t2 = (trace(SUC i))
      in (
        ~((get_trace_agent t2) = SYS_F) ==>
        let ss1 = (get_trace_state t1)
        and ss2 = (get_trace_state t2)
        in (
% no state change %
          (ss1 = ss2) \/
% a message arrives in the F mailbox %
          (F_env_SEND ss1 ss2) \/
% any other state change that does not affect F %
          (F_env_arb ss1 ss2)
        )
      )
    )
  )"
);;




% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define F server state transition relation
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% ================================================================
*
*       F server state transition relation (safety properties)
*

These functions and predicates define the atomic state transition
relation (safety properties) for the F server.

These transitions consist of the F server reading a message from
its mailbox and sending a response message.
```

```
* =================================================================%


% ----------------------------------------------------------------
* F_reads_msg

F reads a message from its mailbox and puts it in the work buffer.

F_reads_msg takes a global state (mailboxes) and F internal state
as input.

Its output is a new system state where the next readable message
in the F system mailbox has been read and placed into the "work"
buffer and the "next" pointer of the F mailbox has been advanced.
If there is no unread message in the mailbox, or if there is
already a message in the work buffer, the input system state is
returned unchanged.

--------------------------------------------------------------- %


let F_reads_msg = new_definition (`F_reads_msg`, "
  (! ss .
  F_reads_msg ss =
  let (mbxs, Fs) = ((get_trace_mbxs ss), (get_trace_Fs ss))
  in
  let mbx = (get_mbxs_mbx mbxs F_ID)
  in
% Check if no messages or if message already in work buffer.
  If so, return state unchanged.                          %
    (((~mbx_is_unread_msg mbx) \/ (Fs_get_flag Fs)) => ss |
% Otherwise, read message... %
    (let (msg, new_mbx) = (read_mbx_msg mbx)
    in
% put back the modified mailbox %
    let new_mbxs = (put_mbxs_mbx mbxs F_ID new_mbx)
% put msg in work buffer %
    and new_Fs = (Fs_put_buf Fs msg)
    in
      (put_trace_mbxs (put_trace_Fs ss new_Fs) new_mbxs)
    ))
  )
");;
```

```
% ------------------------------------------------------------------
* F_reads

True if the second system state is reached from the first due to F
reading a message from its mailbox.

---------------------------------------------------------------- %

let F_reads = new_definition (`F_reads`, "
  (! ss1 ss2 .
  F_reads ss1 ss2 =
    (ss2 = (F_reads_msg ss1))
  )
");;


% ------------------------------------------------------------------
* F_responds_msg

F responds to the message in its "work" buffer.

F_responds_msg takes a system state as input.

Its output is a new system state where the response message has
been sent and the work buffer emptied. If there is no message
already in the work buffer, the input system state is returned
unchanged.

Response messages are sent to the source of the message in the
work buffer (i.e., mbox ids = task ids). The message id is the
same as that of the request message, and the message data is a
function of the data in the request message.

---------------------------------------------------------------- %

let F_responds_msg = new_definition (`F_responds_msg`, "
  (! ss (F_func:num->num).
  F_responds_msg ss F_func =
  let (mbxs, Fs) = ((get_trace_mbxs ss), (get_trace_Fs ss))
  in
% Check if work buffer has a message.
  If not, return state unchanged.             %
    ((~Fs_get_flag Fs) => ss |
% Otherwise, create response message... %
```

```
    (let (rqst, new_Fs) = (Fs_get_buf Fs) % new_Fs has empty buffer %
    in
    let src = (get_msg_sndr rqst) and mdata = (get_msg_data rqst)
    and mid = (get_msg_id rqst)
    in
    let response = (cons_msg F_ID (F_func mdata) mid)
    in
% put message in destination mailbox %
    let new_mbxs = (put_mbxs_msg mbxs src response)
    in
      (put_trace_mbxs (put_trace_Fs ss new_Fs) new_mbxs)
    ))
  )
");;

% ---------------------------------------------------------------
* F_responds

True if the second system state is reached from the first due to F
responding to a message in its work buffer.

--------------------------------------------------------------- %

let F_responds = new_definition (`F_responds`, "
  (! ss1 ss2 (F_func:num->num).
  F_responds ss1 ss2 F_func =
    (ss2 = (F_responds_msg ss1 F_func))
  )
");;


% ---------------------------------------------------------------
* F_safety

In english:

For all elements of the trace, t1 and t2, where t2 is the
immediate successor to t1 and where the agent in t2 is F
  the states in t1 and t2 must be identical
  or the state transition must satisfy the safety
  properties for one of the possible state transitions

Valid state transitions for F are as follows:
```

```
1) Reading a message from the F system mailbox and sending a
response message to the sender.

---------------------------------------------------------------- %

let F_safety = new_definition (`F_safety`, "
  (! (trace : trace_def) (F_func:num->num).
  F_safety trace F_func =
    (! (i : num) .
      let t1 = (trace i)
      and t2 = (trace(SUC i))
      in (
        ((get_trace_agent t2) = SYS_F) ==>
        let ss1 = (get_trace_state t1)
        and ss2 = (get_trace_state t2)
        in (
% no state change %
          (((get_trace_tlabel t2) = STUTTER) /\ (ss1 = ss2)) \/
% or a legal transition %
          (((get_trace_tlabel t2) = FREADS) /\ (F_reads ss1 ss2)) \/
          (((get_trace_tlabel t2) = FRESPONDS) /\
           (F_responds ss1 ss2 F_func))
        )
      )
    )
  )"
);;



% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define F server progress properties
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% ================================================================
*
*       F server state progress properties
*

These properties assert that the F server will eventually read all
messages and send responses.

Assuming that the work buffer is eventually emptied, performing a
F_read operation will read the next unread message in F's mailbox.
```

```
F_reads_progress guarantees that the read will eventually happen.
An (eventually) empty work buffer is guaranteed by the progress
property on the F_responds operation. In this way we guarantee
that all input messages (requests) will eventually be read and
generate responses.

This method of specifying progress properties is possible because
of the way we wrote the safety properties. They were written to
only cause a state change when conditions were correct so that the
operation could be performed. Otherwise, performing either
operation causes no state change. Thus, our progress properties
specify "busy-waiting" on the part of the server.

This method is essentially that of UNITY.

* ==================================================================%


% -------------------------------------------------------------------
* F_reads_progress

A progress property for the ``F'' system call that asserts that,
starting from any point in the trace, F will eventually perform an
F_reads operation.

In english:

For all ``i'' (viz., a trace_element in a trace)
  there is some ``j'' (also a trace_element in the trace)
    such that j is later in the trace than i
    and whose agent is SYS_F (i.e., the agent responsible for the
                                transition)
    and the reason for the state transition is
      a F_reads_msg step
    and in all intermediate steps between i and j
      a F_reads_msg step did not occur.

------------------------------------------------------------------ %

let F_reads_progress = new_definition (`F_reads_progress`,
  "(! (trace : trace_def) .
    F_reads_progress trace =
      (! (i : num) .
        (? (j : num) .
          (i <= j) /\
          (get_trace_state(trace (SUC j)) =
           (F_reads_msg (get_trace_state(trace j))))
        )
      )
   )"
);;



% ------------------------------------------------------------------
* F_responds_progress

A progress property for the ''F'' system call that asserts that,
starting from any point in the trace, F will eventually perform an
F_responds operation.

In english:

For all ''i'' (viz., a trace_element in a trace)
  there is some ''j'' (also a trace_element in the trace)
    such that j is later in the trace than i
    and whose agent is SYS_F (i.e., the agent responsible for the
                                transition)
    and the reason for the state transition is
      a F_responds_msg step
    and in all intermediate steps between i and j
      a F_responds_msg step did not occur.

------------------------------------------------------------------ %

let F_responds_progress = new_definition ('F_responds_progress',
  "(! (trace : trace_def) (F_func:num->num).
    F_responds_progress trace F_func =
      (! (i : num) .
        (? (j : num) .
          (i <= j) /\
          (get_trace_state(trace (SUC j)) =
           (F_responds_msg (get_trace_state(trace j)) F_func))
        )
      )
  )"
);;
```

```
% ------------------------------------------------------------------
* F_progress

The progress properties for the F system call.

------------------------------------------------------------------ %
%
let F_progress = new_definition (`F_progress`, "
  (! (trace : trace_def) (F_func:num->num).
  F_progress trace =
    (F_reads_progress trace) /\
    (F_responds_progress trace F_func)
  )
");;
%
% ------------------------------------------------------------------
* F_env_progress

The progress properties for the F system call environment.

There are no progress properties defined because the server is
purely reactive, so this property is always TRUE.

------------------------------------------------------------------ %

let F_env_progress = new_definition (`F_env_progress`, "
  (! (trace : trace_def) .
  F_env_progress trace =
    T
  )
");;

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define F initial state
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% =================================================================
*
*       initial state
*


*       environmental requirements for initial system state
```

```
The requirements on the system state consist of structural
consitency requirements on the data, which must remain invariant
in every state in a trace, and initial policy conditions (e.g.,
that mailboxes are initially empty).

The environment does not specify any initial conditions on the F
server's internal state.

*       F's requirements for initial internal state

F's requirements for its initial internal state are that its work
buffer is empty.

* ===============================================================%

% ----------------------------------------------------------------
* F_good_init_mbxs

The environmental requirements on the intial system state that are
due to system policy.

We require that all mailboxes are initially empty. We require this
as a convenience, since it is easier than, for example, specifying
the relationship between request and response messages in the
initial state.

---------------------------------------------------------------- %

let F_good_init_mbxs = new_definition (`F_good_init_mbxs`, "
  (! (mbxs : mboxes) .
  F_good_init_mbxs mbxs =
    (! mbx . (mbx IN (rng mbxs)) ==>
     (~mbx_is_unread_msg mbx)
    )
  )
");;

% ----------------------------------------------------------------
* F_env_init

The complete definition of a good initial system state.

The initial system state is correct if it is self-conistent and if
```

```
it satisfies all policy requirements.

The consistency of the system state, once established for the
initial state, must be preserved by all state transitions as an
invariant (i.e., we have to prove that it is invariant).

--------------------------------------------------------------- %

let F_env_init = new_definition (`F_env_init`, "
  (! (trace : trace_def) .
  F_env_init trace =
% require at least one trace element so that we have an initial state %
%    ((LENGTH trace) > 0) /\
    let init_t = (EL 0 trace)%
    let init_t = (trace 0)
    in
    let mbxs = (get_trace_mbxs (get_trace_state init_t))
    in (
      (F_env_invariant mbxs) /\
% self-consistency %
      (good_mbxs mbxs) /\
% policy req's %
      (F_good_init_mbxs mbxs)
    )
  )
");;

% ---------------------------------------------------------------
* F_init

The definition of a good internal state for the F server.

The work buffer must be empty (i.e., there is no response
pending).

F cannot specify any initial conditions on the mailboxes, whose
initial condition is reserved for the environment.

--------------------------------------------------------------- %

let F_init = new_definition (`F_init`, "
  (! (trace : trace_def) .
  F_init trace =
% no need to require at least one trace element because already
```

```
    specified by F_env_good_trace. %
%  let init_t = (EL 0 trace)%
    let init_t = (trace 0)
    in
    let Fs = (get_trace_Fs (get_trace_state init_t))
    in (
    (~Fs_get_flag Fs)
    )
  )
");;

% ------------------------------------------------------------------
* F_system_specification

------------------------------------------------------------------ %
%
let F_system_specification = new_definition
(`F_system_specification`, "
  F_system_specification =
    { trace : trace_def |
      ((F_env_init trace) /\
      (F_env_safety trace) /\
      (F_env_progress trace)) ==>
          ((F_init trace) /\
          (F_safety trace) /\
          (F_progress trace))
    }
");;
%

close_theory();;
```

## A.7 OS-env.ml

```
%-------------------------------------
* File:    OS-env.ml
* Version: 0.0
* Date:    10/23/96

961023. Added transition labels.
*-------------------------------------%
```

```
% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Specification of the OS environment.

Use this for proving composition of F and FG.

The environment is all the other processes.

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

loadt `aux/init.ml`;; new_theory_safe `OS-env`;; load_library
`sets`;; load_parent `F-FG`;;

% ----------------------------------------------------------------
* OS_env_invariant

The conditions of the system state, once established for the
initial state, that must be preserved by all state transitions as
an invariant

---------------------------------------------------------------- %

let OS_env_invariant = new_definition (`OS_env_invariant`, "
  (! mbxs : mboxes .
  OS_env_invariant mbxs =
% F server must have mailbox %
      (F_ID IN (dom mbxs)) /\
% FG server must have mailbox %
      (FG_ID IN (dom mbxs))
  )
");;

% ----------------------------------------------------------------
* OS_env_SEND_F

Returns TRUE if the cause of the state transition between ss1 and
ss2 is a message arriving in the F mailbox. The message must have
a sender id other than F_ID or FG_ID, because the OS did not send
the message.

---------------------------------------------------------------- %

let OS_env_SEND_F = new_definition (`OS_env_SEND_F`, "
  (! (ss1 ss2 : trace_state) .
    OS_env_SEND_F ss1 ss2 =
      let (mbxs1, mbxs2) = ((get_trace_mbxs ss1), (get_trace_mbxs ss2))
      and internals1 = (get_trace_not_mbxs ss1)
      and internals2 = (get_trace_not_mbxs ss2)
      in
      let Fmbx1 = (get_mbxs_mbx mbxs1 F_ID)
      and Fmbx2 = (get_mbxs_mbx mbxs2 F_ID)
      and FGmbx1 = (get_mbxs_mbx mbxs1 FG_ID)
      and FGmbx2 = (get_mbxs_mbx mbxs2 FG_ID)
      in (
        ((dom mbxs1) = (dom mbxs2))  /\
        (internals1 = internals2) /\
        (FGmbx1 = FGmbx2) /\
        (? msg .
          let sndr = (get_msg_sndr msg)
          in (
            (sndr IN (dom mbxs1)) /\
            ~(sndr = F_ID) /\
            ~(sndr = FG_ID) /\
            ((put_mbx_msg Fmbx1 msg) = Fmbx2)
          )
        )
      )
    )
  ");;

% ---------------------------------------------------------------
* OS_env_SEND_FG

Returns TRUE if the cause of the state transition between ss1 and
ss2 is a message arriving in the FG mailbox. The message must have
a sender id other than F_ID or FG_ID, because the OS did not send
the message.

--------------------------------------------------------------- %

let OS_env_SEND_FG = new_definition (`OS_env_SEND_FG`, "
  (! (ss1 ss2 : trace_state) .
   OS_env_SEND_FG ss1 ss2 =
     let (mbxs1, mbxs2) = ((get_trace_mbxs ss1), (get_trace_mbxs ss2))
     and internals1 = (get_trace_not_mbxs ss1)
     and internals2 = (get_trace_not_mbxs ss2)
     in
     let FGmbx1 = (get_mbxs_mbx mbxs1 FG_ID)
    and FGmbx2 = (get_mbxs_mbx mbxs2 FG_ID)
    and Fmbx1 = (get_mbxs_mbx mbxs1 F_ID)
    and Fmbx2 = (get_mbxs_mbx mbxs2 F_ID)
    in (
      ((dom mbxs1) = (dom mbxs2))  /\
      (internals1 = internals2) /\
      (Fmbx1 = Fmbx2) /\
      (? msg .
        let sndr = (get_msg_sndr msg)
        in (
          (sndr IN (dom mbxs1)) /\
          ~(sndr = F_ID) /\
          ~(sndr = FG_ID) /\
          ((put_mbx_msg FGmbx1 msg) = FGmbx2)
        )
      )
    )
  )
");;

% ------------------------------------------------------------------
* OS_env_arb

Returns TRUE if the cause of the state transition between ss1 and
ss2 is a state change to any state components other than the OS
mailboxes, the internal state, and the domain of the mailboxes.

------------------------------------------------------------------ %

let OS_env_arb = new_definition (`OS_env_arb`, "
  (! (ss1 ss2 : trace_state) .
  OS_env_arb ss1 ss2 =
    let (mbxs1, mbxs2) = ((get_trace_mbxs ss1), (get_trace_mbxs ss2))
    and internals1 = (get_trace_not_mbxs ss1)
    and internals2 = (get_trace_not_mbxs ss2)
    in
    let Fmbx1 = (get_mbxs_mbx mbxs1 F_ID)
    and Fmbx2 = (get_mbxs_mbx mbxs2 F_ID)
    and FGmbx1 = (get_mbxs_mbx mbxs1 FG_ID)
    and FGmbx2 = (get_mbxs_mbx mbxs2 FG_ID)
    in (
      ((dom mbxs1) = (dom mbxs2))  /\
      (internals1 = internals2) /\
      (Fmbx1 = Fmbx2) /\
      (FGmbx1 = FGmbx2)
    )
  )
");;


% -----------------------------------------------------------------
* OS_env_safety

In english:

For all elements of the trace, t1 and t2, where t2 is the
immediate successor to t1 and where the agent in t2 is not an OS
agent
  the states in t1 and t2 must be identical
  or the state transition must satisfy the safety
  properties for one of the possible state transitions

Valid state transitions for the OS environment are as follows:

1) A message arriving in a system mailbox.

2) A message being read from a non-system mbox.

No environment state transition may affect the internal state of
the OS.

----------------------------------------------------------------- %

let OS_env_safety = new_definition (`OS_env_safety`,
  "(! (trace : trace_def).
  OS_env_safety trace =
    (! (i : num).
      let t1 = (trace i)
      and t2 = (trace(SUC i))
      in (
        (~(get_trace_agent t2 = SYS_F) /\
        ~(get_trace_agent t2 = SYS_FG)) ==>
        let ss1 = (get_trace_state t1)
        and ss2 = (get_trace_state t2)
        in (
          (ss1 = ss2) \/
          (OS_env_SEND_F ss1 ss2) \/
          (OS_env_SEND_FG ss1 ss2) \/
          (OS_env_arb ss1 ss2)
         )
        )
       )
  )"
);;


% ---------------------------------------------------------------
* OS_env_progress

There are no OS environment progress properties.

--------------------------------------------------------------- %

let OS_env_progress = new_definition (`OS_env_progress`, "
  (! (trace : trace_def) .
  OS_env_progress trace =
    T
  )
");;

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define initial state
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% ===============================================================
*
*       initial state
*


*       environmental requirements for initial system state

The requirements on the system state consist of structural
consitency requirements on the data, which must remain invariant
in every state in a trace, and initial policy conditions (e.g.,
that mailboxes are initially empty).

The environment does not specify any initial conditions on the
OS's internal state.

* ==============================================================%

% ---------------------------------------------------------------
```

```
* OS_good_init_mbxs

The environmental requirements on the intial system state that are
due to system policy.

We require that all mailboxes are initially empty. We require this
as a convenience, since it is easier than, for example, specifying
the relationship between request and response messages in the
initial state.

------------------------------------------------------------- %

let OS_good_init_mbxs = new_definition (`OS_good_init_mbxs`, "
  (! (mbxs : mboxes) .
  OS_good_init_mbxs mbxs =
    (! mbx . (mbx IN (rng mbxs)) ==>
      (~mbx_is_unread_msg mbx)
    )
  )
");;


% -------------------------------------------------------------
* OS_env_init

The complete definition of a good initial system state.

The initial system state is correct if it is self-conistent and if
it satisfies all policy requirements.

The consistency of the system state, once established for the
initial state, must be preserved by all state transitions as an
invariant (i.e., we have to prove that it is invariant).

------------------------------------------------------------- %

let OS_env_init = new_definition (`OS_env_init`, "
  (! (trace : trace_def) .
  OS_env_init trace =
% need unique mailboxes for the servers %
    ~(F_ID = FG_ID) /\
    let init_t = (trace 0)
    in
    let mbxs = (get_trace_mbxs (get_trace_state init_t))
    in (
      (OS_env_invariant mbxs) /\
% self-consistency %
      (good_mbxs mbxs) /\
% policy req's %
      (OS_good_init_mbxs mbxs)
    )
  )
");;


close_theory();;
```

## A.8 OS-sys.ml

```
%-------------------------------------
* File:    OS-sys.ml
* Version: 0.0
* Date:    05/20/98

Replaced FG_func with universally quantified function.
*-------------------------------------%

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Specification of the OS system.

The OS in this example consists of the F and FG system calls. The
specification says that the OS will read messages from its F and
FG mailboxes and eventually send responses back, in the order that
the requests were sent.

The environment is all the other processes, and is defined in
OS-env.ml.

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

loadt `aux/init.ml`;;

new_theory_safe `OS-sys`;;

load_library `sets`;;
```

```
load_parent `silobasic`;; load_parent `F-FG`;;

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define initial state
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% ================================================================
server ids

The mailbox ids of the OS mailboxes. These are defined in F-FG.ml.
================================================================ %

% ================================================================
    OS internal state

The internal state of the OS.

The initial system state is defined by the environment
(OS-env.ml). The environment does not specify any initial
conditions on the OS's internal state.

The internal state is modifiable only by the OS in the
specification, and not by the environment.

There is a different internal state for each system call that
consists of a message queue. The queue is used to store messages
between the atomic actions of reading a message from the input
mailbox and of sending a response.

================================================================ %

new_type_abbrev (`msg_list`, ":(msg_def)list");; new_type_abbrev
(`OSF_queue`, ":msg_list");; new_type_abbrev (`OSFG_queue`,
":msg_list");;

new_type_abbrev (`OSF_internal_state`,":OSF_queue");;
new_type_abbrev (`OSFG_internal_state`,":OSFG_queue");;

new_type_abbrev
(`OS_internal_state`,":OSF_internal_state#OSFG_internal_state");;

% ----------------------------------------------------------------
    FG state utility functions
```

```
----------------------------------------------------------------- %

% -----------------------------------------------------------------
* OSis_get_Fis

Extracts the F internal state from an OSis internal state.

----------------------------------------------------------------- %

let OSis_get_Fis = new_definition (`OSis_get_Fis`, "
  (! (OSis : OS_internal_state).
    OSis_get_Fis OSis = (FST OSis)
  )
");;

% -----------------------------------------------------------------
* OSis_get_FGis

Extracts the FG internal state from an OSis internal state.

----------------------------------------------------------------- %

let OSis_get_FGis = new_definition (`OSis_get_FGis`, "
  (! (OSis : OS_internal_state).
    OSis_get_FGis OSis = (SND OSis)
  )
");;

% -----------------------------------------------------------------
* OSis_put_Fis

Replaces the F internal state in an OSis internal state.

----------------------------------------------------------------- %

let OSis_put_Fis = new_definition (`OSis_put_Fis`, "
  (! (OSis : OS_internal_state) (newFis : OSF_internal_state).
    OSis_put_Fis OSis newFis =
      let FGis = (OSis_get_FGis OSis) in
      (newFis,FGis)
  )
");;

% -----------------------------------------------------------------
```

```
* OSis_put_FGis

Replaces the FG internal state in an OSis internal state.

---------------------------------------------------------------- %

let OSis_put_FGis = new_definition (`OSis_put_FGis`, "
  (! (OSis : OS_internal_state) (newFGis : OSFG_internal_state).
    OSis_put_FGis OSis newFGis =
      let Fis = OSis_get_Fis OSis in
      (Fis,newFGis)
  )
");;

% ----------------------------------------------------------------
* OSis_get_Fq

Extracts the F internal queue from an OSis internal state.

---------------------------------------------------------------- %

let OSis_get_Fq = new_definition (`OSis_get_Fq`, "
  (! (OSis : OS_internal_state).
    OSis_get_Fq OSis = (OSis_get_Fis OSis)
  )
");;

% ----------------------------------------------------------------
* OSis_get_FGq

Extracts the FG internal queue from an OSis internal state.

---------------------------------------------------------------- %

let OSis_get_FGq = new_definition (`OSis_get_FGq`, "
  (! (OSis : OS_internal_state).
    OSis_get_FGq OSis = (OSis_get_FGis OSis)
  )
");;

% ----------------------------------------------------------------
* OSFis_put_Fq

Replaces the F internal queue in an OSF internal state.
```

```
---------------------------------------------------------------- %

let OSFis_put_Fq = new_definition (`OSFis_put_Fq`, "
  (! (Fis : OSF_internal_state) (newFq : OSF_queue).
    OSFis_put_Fq Fis newFq = (newFq)
  )
");;

% ----------------------------------------------------------------
* OSFGis_put_FGq

Replaces the FG internal queue in an OSFG internal state.

---------------------------------------------------------------- %

let OSFGis_put_FGq = new_definition (`OSFGis_put_FGq`, "
  (! (FGis : OSFG_internal_state) (newFGq : OSFG_queue).
    OSFGis_put_FGq FGis newFGq = (newFGq)
  )
");;

% ----------------------------------------------------------------
* OSis_put_Fq

Replaces the F internal queue in an OSis internal state.

---------------------------------------------------------------- %

let OSis_put_Fq = new_definition (`OSis_put_Fq`, "
  (! (OSis : OS_internal_state) (newFq : OSF_queue).
    OSis_put_Fq OSis newFq = (
    let oldFis = (OSis_get_Fis OSis) in
      OSis_put_Fis OSis (OSFis_put_Fq oldFis newFq)
    )
  )
");;

% ----------------------------------------------------------------
* OSis_put_FGq

Replaces the FG internal queue in an OSis internal state.

---------------------------------------------------------------- %
```

```
let OSis_put_FGq = new_definition (`OSis_put_FGq`, "
  (! (OSis : OS_internal_state) (newFGq : OSFG_queue).
    OSis_put_FGq OSis newFGq = (
    let oldFGis = (OSis_get_FGis OSis) in
      OSis_put_FGis OSis (OSFGis_put_FGq oldFGis newFGq)
    )
  )
");;

% ----------------------------------------------------------------
* OSis_put_Fq_msg

Adds a message to the end of the F internal queue in an OSis
internal state.

---------------------------------------------------------------- %

let OSis_put_Fq_msg = new_definition (`OSis_put_Fq_msg`, "
  (! (OSis : OS_internal_state) (msg : msg_def).
    OSis_put_Fq_msg OSis msg = (
    let oldFq = (OSis_get_Fq OSis)
    in
      (OSis_put_Fq
        OSis
        (SNOC msg oldFq)
      )
    )
  )
");;

% ----------------------------------------------------------------
* OSis_put_FGq_msg

Adds a message to the end of the FG internal queue in an OSis
internal state.

---------------------------------------------------------------- %

let OSis_put_FGq_msg = new_definition (`OSis_put_FGq_msg`, "
  (! (OSis : OS_internal_state) (msg : msg_def).
    OSis_put_FGq_msg OSis msg = (
    let oldFGq = (OSis_get_FGq OSis)
    in
```

```
        (OSis_put_FGq
          OSis
          (SNOC msg oldFGq)
        )
      )
    )
");;

% ---------------------------------------------------------------
* OSis_Fq_pending

TRUE if there is at least one unread msg in the F queue.

--------------------------------------------------------------- %

let OSis_Fq_pending = new_definition (`OSis_Fq_pending`, "
  (! (OSis : OS_internal_state) .
    OSis_Fq_pending OSis =
    let Fq = (OSis_get_Fq OSis)
    in
      (0 < (LENGTH Fq))
  )
");;


% ---------------------------------------------------------------
* OSis_FGq_pending

TRUE if there is at least one unread msg in the FG queue.

--------------------------------------------------------------- %

let OSis_FGq_pending = new_definition (`OSis_FGq_pending`, "
  (! (OSis : OS_internal_state) .
    OSis_FGq_pending OSis =
    let FGq = (OSis_get_FGq OSis)
    in
      (0 < (LENGTH FGq))
  )
");;

% ---------------------------------------------------------------
* OSis_Fq_get_msg
```

```
Returns the next unread message in the queue. Automatically
removes the message from the queue and returns the updated OSis
state along with the msg.

UNDEFINED if there are no unread messages. Safe if used with
``OSis_Fq_pending'', as follows:

(OSis_Fq_pending OSis) => (OSis_Fq_get_msg OSis) | ...

---------------------------------------------------------------- %

let OSis_Fq_get_msg = new_definition (`OSis_Fq_get_msg`, "
  (! (OSis : OS_internal_state) .
    OSis_Fq_get_msg OSis =
    let Fq = (OSis_get_Fq OSis)
    in
    let msg = (HD Fq)
    and newFq = (TL Fq)
    in
      (msg, (OSis_put_Fq OSis newFq))
  )
");;

% ----------------------------------------------------------------
* OSis_FGq_get_msg

Returns the next unread message in the queue. Automatically
removes the message from the queue and returns the updated OSis
state along with the msg.

UNDEFGINED if there are no unread messages. Safe if used with
``OSis_FGq_pending'', as follows:

(OSis_FGq_pending OSis) => (OSis_FGq_get_msg OSis) | ...

---------------------------------------------------------------- %

let OSis_FGq_get_msg = new_definition (`OSis_FGq_get_msg`, "
  (! (OSis : OS_internal_state) .
    OSis_FGq_get_msg OSis =
    let FGq = (OSis_get_FGq OSis)
    in
    let msg = (HD FGq)
    and newFGq = (TL FGq)
```

```
     in
       (msg, (OSis_put_FGq OSis newFGq))
   )
");;


new_type_abbrev ('OS_trace_state', ":mboxes#OS_internal_state");;
new_type_abbrev ('OS_trace_element',
":Agent#TLabel#OS_trace_state");; new_type_abbrev ('OS_trace_def',
":num->OS_trace_element");;

% ------------------------------------------------------------------
    trace utility functions

------------------------------------------------------------------ %

% ------------------------------------------------------------------
* get_OStrace_agent

------------------------------------------------------------------ %

let get_OStrace_agent = new_definition ('get_OStrace_agent', "
  (! (tel : OS_trace_element).
    get_OStrace_agent tel = (FST tel)
  )
");;

% ------------------------------------------------------------------
* get_OStrace_tlabel

------------------------------------------------------------------ %

let get_OStrace_tlabel = new_definition ('get_OStrace_tlabel', "
  (! (tel : OS_trace_element).
    get_OStrace_tlabel tel = FST(SND tel)
  )
");;

% ------------------------------------------------------------------
* get_OStrace_state

------------------------------------------------------------------ %

let get_OStrace_state = new_definition ('get_OStrace_state', "
  (! (tel : OS_trace_element).
    get_OStrace_state tel = SND(SND tel)
  )
");;

% ---------------------------------------------------------------
* get_OStrace_mbxs

--------------------------------------------------------------- %

let get_OStrace_mbxs = new_definition (`get_OStrace_mbxs`, "
  (! (ts : OS_trace_state).
    get_OStrace_mbxs ts = (FST ts)
  )
");;


% ---------------------------------------------------------------
* get_OStrace_not_mbxs

--------------------------------------------------------------- %

let get_OStrace_not_mbxs = new_definition (`get_OStrace_not_mbxs`,
"
  (! (ts : OS_trace_state).
    get_OStrace_not_mbxs ts = (SND ts)
  )
");;

% ---------------------------------------------------------------
* get_OStrace_is

--------------------------------------------------------------- %

let get_OStrace_is = new_definition (`get_OStrace_is`, "
  (! (ts : OS_trace_state).
    get_OStrace_is ts = (SND ts)
  )
");;

% ---------------------------------------------------------------
* put_OStrace_mbxs

--------------------------------------------------------------- %
```

```
let put_OStrace_mbxs = new_definition (`put_OStrace_mbxs`, "
  (! (ts : OS_trace_state) (new_mbxs : mboxes) .
    put_OStrace_mbxs ts new_mbxs =
    let (old_mbxs, OSis) = ts
    in
      (new_mbxs, OSis)
  )
");;

% ----------------------------------------------------------------
* put_OStrace_is

---------------------------------------------------------------- %

let put_OStrace_is = new_definition (`put_OStrace_is`, "
  (! (ts : OS_trace_state) (new_OSis : OS_internal_state) .
    put_OStrace_is ts new_OSis =
    let (mbxs, old_OSis) = ts
    in
      (mbxs, new_OSis)
  )
");;


% ----------------------------------------------------------------
* get_OStrace_Fis

---------------------------------------------------------------- %

let get_OStrace_Fis = new_definition (`get_OStrace_Fis`, "
  (! (ts : OS_trace_state).
    get_OStrace_Fis ts = (OSis_get_Fis (get_OStrace_is ts))
  )
");;

% ----------------------------------------------------------------
* get_OStrace_FGis

---------------------------------------------------------------- %

let get_OStrace_FGis = new_definition (`get_OStrace_FGis`, "
  (! (ts : OS_trace_state).
    get_OStrace_FGis ts = (OSis_get_FGis (get_OStrace_is ts))
  )
");;

% -------------------------------------------------------------
* put_OStrace_Fis

------------------------------------------------------------- %

let put_OStrace_Fis = new_definition (`put_OStrace_Fis`, "
  (! (ts : OS_trace_state) (new_Fis : OSF_internal_state) .
    put_OStrace_Fis ts new_Fis =
    let old_is = (get_OStrace_is ts)
    in
    let new_is = (OSis_put_Fis old_is new_Fis)
    in
      (put_OStrace_is ts new_is)
  )
");;


% -------------------------------------------------------------
* put_OStrace_FGis

------------------------------------------------------------- %

let put_OStrace_FGis = new_definition (`put_OStrace_FGis`, "
  (! (ts : OS_trace_state) (new_FGis : OSFG_internal_state) .
    put_OStrace_FGis ts new_FGis =
    let old_is = (get_OStrace_is ts)
    in
    let new_is = (OSis_put_FGis old_is new_FGis)
    in
      (put_OStrace_is ts new_is)
  )
");;


% -------------------------------------------------------------
* cons_OStrace_state

------------------------------------------------------------- %

let cons_OStrace_state = new_definition (`cons_OStrace_state`, "
  (! (newmbxs:mboxes)
      (newFis:OSF_internal_state)
      (newFGis : OSFG_internal_state) .
     cons_OStrace_state newmbxs newFis newFGis =
       (newmbxs,(newFis,newFGis))
  )
");;


% ------------------------------------------------------------------
* cons_OStrace_el

------------------------------------------------------------------ %

let cons_OStrace_el = new_definition (`cons_OStrace_el`, "
  (! (tagent:Agent) (tlabel:TLabel) (tstate:OS_trace_state) .
    cons_OStrace_el tagent tlabel tstate =
      (tagent,tlabel,tstate)
  )
");;



% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define environment state transition relation
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% =================================================================
*
*       environment state transition relation (safety properties)
*

These functions and predicates define the state transition
relation (safety properties) for OS environment, which consists of
tasks other than the servers.

The environment state transition relation is defined in the file
OS-env.ml.

* ================================================================%


% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define OS state transition relation
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %
```

```
% ================================================================
*
*       OS state transition relation (safety properties)
*

These functions and predicates define the atomic state transition
relation (safety properties) for the OS.

These transitions consist of the OS reading a message from one of
its mailboxes and sending response messages. The OS can either
read one message from one of its mailboxes or it can respond to a
message that it has already read.

We specify the OS as having unlimited buffering capability to
store messages that have been read but for which responses have
not yet been sent, but do not require this. A progress property
insures that responses will eventually be sent. An implementation
will realistically have only limited buffering capacity, but,
because of the progress property, any amount of buffering, even
none, can satisfy the specification.

* ================================================================%


% ----------------------------------------------------------------
* OS_reads_F_msg

OS reads a message from its F mailbox and puts it in the F queue.

OS_reads_F_msg takes a trace state as input.

Its output is a new state where the next readable message in the F
system mailbox has been read and placed into the corresponding
queue and the "next" pointer of the mailbox has been advanced. If
there is no unread message in the mailbox the input state is
returned unchanged.

---------------------------------------------------------------- %

let OS_reads_F_msg = new_definition (`OS_reads_F_msg`, "
  (! ss .
  OS_reads_F_msg ss =
  let mbxs = (get_OStrace_mbxs ss)
  and OSis = (get_OStrace_is ss)
  in
  let Fmbx = (get_mbxs_mbx mbxs F_ID)
  in
% Check if no messages. If so, return state unchanged. %
    ((~mbx_is_unread_msg Fmbx) => ss |
% Otherwise, read message... %
    (let (msg, new_Fmbx) = (read_mbx_msg Fmbx)
    in
% put back the modified mailbox %
    let new_mbxs = (put_mbxs_mbx mbxs F_ID new_Fmbx)
% put msg in queue %
    and new_OSis = (OSis_put_Fq_msg OSis msg)
    in
      (put_OStrace_mbxs (put_OStrace_is ss new_OSis) new_mbxs)
    ))
  )
");;


% ----------------------------------------------------------------
* OS_reads_F

True if the second system state is reached from the first due to
OS reading a message from its mailbox.

---------------------------------------------------------------- %

let OS_reads_F = new_definition (`OS_reads_F`, "
  (! ss1 ss2 .
  OS_reads_F ss1 ss2 =
    (ss2 = (OS_reads_F_msg ss1))
  )
");;

% ----------------------------------------------------------------
* OS_reads_FG_msg

OS reads a message from its FG mailbox and puts it in the FG
queue.

OS_reads_FG_msg takes a trace state as input.

Its output is a new state where the next readable message in the
```

```
FG system mailbox has been read and placed into the corresponding
queue and the "next" pointer of the mailbox has been advanced. If
there is no unread message in the mailbox the input state is
returned unchanged.

---------------------------------------------------------------- %

let OS_reads_FG_msg = new_definition (`OS_reads_FG_msg`, "
  (! ss .
  OS_reads_FG_msg ss =
  let mbxs = (get_OStrace_mbxs ss)
  and OSis = (get_OStrace_is ss)
  in
  let FGmbx = (get_mbxs_mbx mbxs FG_ID)
  in
% Check if no messages. If so, return state unchanged. %
    ((~mbx_is_unread_msg FGmbx) => ss |
% Otherwise, read message... %
    (let (msg, new_FGmbx) = (read_mbx_msg FGmbx)
    in
% put back the modified mailbox %
    let new_mbxs = (put_mbxs_mbx mbxs FG_ID new_FGmbx)
% put msg in queue %
    and new_OSis = (OSis_put_FGq_msg OSis msg)
    in
      (put_OStrace_mbxs (put_OStrace_is ss new_OSis) new_mbxs)
    ))
  )
");;

% ----------------------------------------------------------------
* OS_reads_FG

True if the second system state is reached from the first due to
OS reading a message from its mailbox.

---------------------------------------------------------------- %

let OS_reads_FG = new_definition (`OS_reads_FG`, "
  (! ss1 ss2 .
  OS_reads_FG ss1 ss2 =
    (ss2 = (OS_reads_FG_msg ss1))
  )
");;
```

```
% -----------------------------------------------------------------
* OS_responds_F_msg

OS responds to a message in its F queue.

OS_responds_F_msg takes a system state as input.

The output is a new system state where the message at the head of
the queue has been removed, and a response message has been sent.
If there is no message in the queue, the input state is returned
unchanged.

----------------------------------------------------------------- %

let OS_responds_F_msg = new_definition (`OS_responds_F_msg`, "
  (! ss (F_func:num->num).
  OS_responds_F_msg ss F_func =
  let mbxs = (get_OStrace_mbxs ss)
  and OSis = (get_OStrace_is ss)
  in
% Check if F queue has a message.
  If not, return state unchanged. %
    ((~(OSis_Fq_pending OSis)) => ss |
% Otherwise, create response message... %
    (let (rqst, new_OSis) = (OSis_Fq_get_msg OSis)
    in
    let src = (get_msg_sndr rqst) and mdata = (get_msg_data rqst)
    and mid = (get_msg_id rqst)
    in
    let response = (cons_msg F_ID (F_func mdata) mid)
    in
% put message in destination mailbox %
    let new_mbxs = (put_mbxs_msg mbxs src response)
    in
      (put_OStrace_mbxs (put_OStrace_is ss new_OSis) new_mbxs)
    ))
  )
");;


% -----------------------------------------------------------------
* OS_responds_F
```

```
True if the second system state is reached from the first due to
OS responding to a message in its queue.

---------------------------------------------------------------- %

let OS_responds_F = new_definition (`OS_responds_F`, "
  (! ss1 ss2 (F_func:num->num).
  OS_responds_F ss1 ss2 F_func =
    (ss2 = (OS_responds_F_msg ss1 F_func))
  )
");;

% ----------------------------------------------------------------
* OS_responds_FG_msg

OS responds to a message in its FG queue.

OS_responds_FG_msg takes a system state as input.

The output is a new system state where the message at the head of
the queue has been removed, and a response message has been sent.
If there is no message in the queue, the input state is returned
unchanged.

---------------------------------------------------------------- %

let OS_responds_FG_msg = new_definition (`OS_responds_FG_msg`, "
  (! ss (FG_func:num->num).
  OS_responds_FG_msg ss FG_func =
  let mbxs = (get_OStrace_mbxs ss)
  and OSis = (get_OStrace_is ss)
  in
% Check if FG queue has a message.
  If not, return state unchanged.            %
    ((~(OSis_FGq_pending OSis)) => ss |
% Otherwise, create response message... %
    (let (rqst, new_OSis) = (OSis_FGq_get_msg OSis)
    in
    let src = (get_msg_sndr rqst) and mdata = (get_msg_data rqst)
    and mid = (get_msg_id rqst)
    in
    let response = (cons_msg FG_ID (FG_func mdata) mid)
    in
% put message in destination mailbox %
    let new_mbxs = (put_mbxs_msg mbxs src response)
    in
      (put_OStrace_mbxs (put_OStrace_is ss new_OSis) new_mbxs)
    ))
  )
");;

% ----------------------------------------------------------------
* OS_responds_FG

True if the second system state is reached from the first due to
OS responding to a message in its queue.

---------------------------------------------------------------- %

let OS_responds_FG = new_definition (`OS_responds_FG`, "
  (! ss1 ss2 (FG_func:num->num).
  OS_responds_FG ss1 ss2 FG_func =
    (ss2 = (OS_responds_FG_msg ss1 FG_func))
  )
");;


% ----------------------------------------------------------------
* OS_safety

In english:

For all elements of the trace, t1 and t2, where t2 is the
immediate successor to t1 and where the agent in t2 is OS
  the states in t1 and t2 must be identical
  or the state transition must satisfy the safety
  properties for one of the possible state transitions

Valid state transitions for OS are as follows:

Reading request messages from OS system mailboxes and sending
responses.

---------------------------------------------------------------- %

let OS_safety = new_definition (`OS_safety`,
  "(!(trace : OS_trace_def) (F_func:num->num) (FG_func:num->num).
  OS_safety trace F_func FG_func =
    (! (i : num) .
      let t1 = (trace i)
      and t2 = (trace(SUC i))
      in (
        ~(get_OStrace_agent t2 = ENV) ==>
        let ss1 = (get_OStrace_state t1)
        and ss2 = (get_OStrace_state t2)
        in (
% no state change %
          (ss1 = ss2) \/
% or a legal transition %
          ((((get_OStrace_tlabel t2) = FREADS) /\ OS_reads_F ss1 ss2)) \/
          ((((get_OStrace_tlabel t2) = FGREADS) /\
            OS_reads_FG ss1 ss2)) \/
          ((((get_OStrace_tlabel t2) = FRESPONDS) /\
            OS_responds_F ss1 ss2 F_func)) \/
          ((((get_OStrace_tlabel t2) = FGRESPONDS) /\
            OS_responds_FG ss1 ss2 FG_func))
        )
      )
    )
  )"
);;



% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define OS progress properties
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% ================================================================
*
*       OS state progress properties
*

We have written our safety properties so that they are always
enabled, but leave the state unchanged if conditions are not
right. In this way, we simplify writing the progress properties to
merely assert that, at any point in the trace, a transition that
satisfies a particular safety property will eventually occur. This
approach is essentially that of UNITY. Alternatively, we can think
of it as "weak fairness", because steps are guaranteed to be
executed infinitely often, but they are not guaranteed to be
```

```
enabled infinitely often and therefore are not guaranteed to ever
happen, depending on the particular step.
* ================================================================%



% ---------------------------------------------------------------
* OS_reads_F_progress

A progress property that asserts that, starting from any point in
the trace, the OS will eventually perform an OS_reads_F operation
after some intervening series of non-OS_reads_F steps.

In english:

For all ``i'' (viz., a trace_element in a trace)
  there is some ``j'' (also a trace_element in the trace)
    such that j is later in the trace than i
    and whose agent is not ENV (i.e., a system agent is
            responsible for the transition)
    and the reason for the state transition is
      a OS_reads_F_msg step
    and in all intermediate steps between i and j
      a OS_reads_F_msg step did not occur.

--------------------------------------------------------------- %

let OS_reads_F_progress = new_definition (`OS_reads_F_progress`,
  "(! (trace : OS_trace_def) .
    OS_reads_F_progress trace =
      (! (i : num) .
        (? (j : num) .
          (i <= j) /\
          (get_OStrace_state(trace(SUC j)) =
           (OS_reads_F_msg (get_OStrace_state(trace j))))
    )
      )
  )"
);;



% ---------------------------------------------------------------
```

```
* OS_responds_F_progress

A progress property that asserts that, starting from any point in
the trace, the OS will eventually perform an OS_responds_F
operation after some intervening series of non-OS_responds_F
steps.

In english:

For all ``i'' (viz., a trace_element in a trace)
  there is some ``j'' (also a trace_element in the trace)
    such that j is later in the trace than i
    and whose agent is not ENV (i.e., a system agent is
            responsible for the transition)
    and the reason for the state transition is
      a OS_responds_F_msg step
    and in all intermediate steps between i and j
      a OS_responds_F_msg step did not occur.

---------------------------------------------------------------- %

let OS_responds_F_progress =
     new_definition (`OS_responds_F_progress`,
  "(! (trace : OS_trace_def) (F_func:num->num) .
    OS_responds_F_progress trace F_func =
      (! (i : num) .
        (? (j : num) .
          (i <= j) /\
          (get_OStrace_state(trace(SUC j)) =
           (OS_responds_F_msg (get_OStrace_state(trace j)) F_func))
        )
      )
  )"
);;



% ----------------------------------------------------------------
* OS_reads_FG_progress

A progress property that asserts that, starting from any point in
the trace, the OS will eventually perform an OS_reads_FG operation
after some intervening series of non-OS_reads_FG steps.
```

```
In english:

For all ''i'' (viz., a trace_element in a trace)
  there is some ''j'' (also a trace_element in the trace)
    such that j is later in the trace than i
    and whose agent is not ENV (i.e., a system agent is
            responsible for the transition)
    and the reason for the state transition is
      a OS_reads_FG_msg step
    and in all intermediate steps between i and j
      a OS_reads_FG_msg step did not occur.

--------------------------------------------------------------- %

let OS_reads_FG_progress = new_definition ('OS_reads_FG_progress',
  "(! (trace : OS_trace_def) .
    OS_reads_FG_progress trace =
      (! (i : num) .
        (? (j : num) .
          (i <= j) /\
          (get_OStrace_state(trace(SUC j)) =
           (OS_reads_FG_msg (get_OStrace_state(trace j))))
        )
      )
  )"
);;



% ---------------------------------------------------------------
* OS_responds_FG_progress

A progress property that asserts that, starting from any point in
the trace, the OS will eventually perform an OS_responds_FG
operation after some intervening series of non-OS_responds_FG
steps.

In english:

For all ''i'' (viz., a trace_element in a trace)
  there is some ''j'' (also a trace_element in the trace)
    such that j is later in the trace than i
    and whose agent is not ENV (i.e., a system agent is
            responsible for the transition)
    and the reason for the state transition is
      a OS_responds_FG_msg step
    and in all intermediate steps between i and j
      a OS_responds_FG_msg step did not occur.

---------------------------------------------------------------- %

let OS_responds_FG_progress =
     new_definition (`OS_responds_FG_progress`,
  "(! (trace : OS_trace_def) (FG_func:num->num).
    OS_responds_FG_progress trace FG_func =
      (! (i : num) .
        (? (j : num) .
          (i <= j) /\
          (get_OStrace_state(trace(SUC j)) =
           (OS_responds_FG_msg (get_OStrace_state(trace j)) FG_func))
        )
      )
  )"
);;



% ----------------------------------------------------------------
* OS_progress

The progress properties for the OS system calls.

---------------------------------------------------------------- %
%
let OS_progress = new_definition (`OS_progress`, "
  (! (trace : OS_trace_def) .
  OS_progress trace =
    (OS_reads_F_progress trace) /\
    (OS_reads_FG_progress trace) /\
    (OS_responds_F_progress trace) /\
    (OS_responds_FG_progress trace)
  )
");;
%

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    Define initial state
```

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% =================================================================
*
*       initial state
*

The environment defines the initial conditions on the system
state. Only the initial conditions on the OS internal state are
defined here.

The OS's requirements for its initial internal state are that its
queues are empty.

* ================================================================%

% ----------------------------------------------------------------
* OS_init

The definition of a good internal state for the OS.

The queue must be empty (i.e., there are no responses pending) for
all servers.

OS cannot specify any initial conditions on the mailboxes, whose
initial condition is reserved for the environment.

---------------------------------------------------------------- %

let OS_init = new_definition (`OS_init`, "
  (! (trace : OS_trace_def) .
  OS_init trace =
    let init_t = (trace 0)
    in
    let OSis = (get_OStrace_is (get_OStrace_state init_t))
    in (
     ~(OSis_Fq_pending OSis) /\
     ~(OSis_FGq_pending OSis)
    )
  )
");;


close_theory();;
```

## A.9 letconv.ml

```
%------------------------------------
* File:    letconv.ml
* Version: 0.0
* Date:    04/04/96
*------------------------------------%

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Conversions for let statements.

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% Safe matching %

let mustUnify tm1 tm2 =
      (true,(match tm1 tm2)) ? (false,([],[])) ;;

%-------------------------------------------------------------
* genSubstPat and helpers
*
* Generate a term suitable as the pattern to SUBST, given a
* higher-level description of where substitutions are desired.
* (e.g. see convCond, where the pattern "A:* = (X => B | C)" is
* used to generate a pattern for substituting the test expr of
* the conditional ("X marks the spot")).
*-------------------------------------------------------------%

letrec dropx plist =
  ( if (plist = []) then
       []
    else if ((fst(dest_var(snd(hd plist)))) = `X`) then
       (dropx (tl plist))
    else
       ((hd plist) . (dropx (tl plist)))
  )
  ? (failwith `dropx`);;

letrec getx plist =
  ( if (plist = []) then
       (failwith `getx: empty plist`)
    else if ((fst(dest_var(snd(hd plist)))) = `X`) then
       (snd (hd plist))
    else
       (getx (tl plist))
  )
  ? (failwith `getx`);;

let genSubstPat pat data =
  ( let info = mustUnify pat data
    in
        (if (not (fst info)) then
            (failwith `genSubstPat: pat dat do not match`)
         else
            let ipat  = (inst [] (snd(snd info)) pat) in
            let fills = dropx (fst(snd info))
            and xtm   = getx  (fst(snd info))
            in
                ((subst fills ipat),xtm))
  )
  ? (failwith `genSubstPat`);;

let expand_let_pair_CONV tm =
  if (is_let tm) then
     let (bod,exp) = (dest_let tm)
     in
       (if not (is_pair exp) then
         (if (is_pabs bod) then
              let eqThm  = (REFL tm) in
              let subThm =
                  (SYM (ISPEC exp PAIR)) in
              let (pt,xt) =
                  (genSubstPat "B = ^(mk_let("A:*->**","X:*"))"
                               (concl eqThm))
              in
                  (SUBST [subThm,xt] pt eqThm)
          else
              (failwith `expand_let_pair_CONV: simple let`))
        else
          (failwith `expand_let_pair_CONV: already paired`))
  else
     (failwith `expand_let_pair_CONV: not let`) ;;
```

## A.10 siloprojection.ml

```
%-------------------------------------
```

```
* File:    siloprojection.ml
* Version: 0.0
* Date:    02/28/96
*-------------------------------------%

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Definitions and theorems for projections.

All theorems in this library are based on the definition of
"project_f_lst".

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %


loadf `aux/init.ml`;; new_theory_safe `siloprojection`;;
load_parent `silomorelists`;; load_library `arith`;;



% ---------------------------------------------------------------
* project_f_lst

Given a list of type * and a boolean filter function, returns a
list of the elements of the input list that satisfy the filter
function, in their original order.

Can be used to project a list of elements that satisfy the
function (or, using negation, to filter out elements for which the
function is false).
--------------------------------------------------------------- %

let project_f_lst =
  new_recursive_definition false SNOC_Axiom `project_f_lst`
  "(! (f : (*->bool)) .
      (project_f_lst f ([] : (*)list) = ([]: (*)list))
   ) /\
   (! (f : (*->bool)) (d : (*)) (t : (*)list) .
      (project_f_lst f (SNOC d t) =
        ((f d) =>
          (SNOC d (project_f_lst f t))
        | (project_f_lst f t)
        )
      )
   )";;
```

```
let member_projection = prove_thm(`member_projection`,
  "(!(l:(* list)) (f:(*->bool)) (x:*) .
     (MEMBER x (project_f_lst f l) ==> f x)
  )",
  SNOC_INDUCT_TAC THEN
  REPEAT GEN_TAC THEN
  REWRITE_TAC [project_f_lst] THENL [
    REWRITE_TAC [MEMBER]
  ;
    COND_CASES_TAC THENL [
      REWRITE_TAC [MEMBER_SNOC] THEN
      STRIP_TAC THENL [
        sym_ASM_PICK_TAC "(x:*) = x'" THEN
        ASM_REWRITE_TAC []
      ;
        mod_ASM_PICK_TAC
          "!f x. MEMBER x(project_f_lst f (l:(* list))) ==> f x"
          (SPECL ["f:(*->bool)";"x':*"]) THEN
        RES_TAC
      ]
    ;
      ASM_REWRITE_TAC []
    ]
  ]
);;


let projection_member = prove_thm(`projection_member`,
  "(!(l:(* list)) (f:(*->bool)) (x:*) .
     ((f x) /\ (MEMBER x l)) ==>
     (MEMBER x (project_f_lst f l))
  )",
  SNOC_INDUCT_TAC THEN
  REPEAT GEN_TAC THEN
  REWRITE_TAC [project_f_lst] THENL [
    REWRITE_TAC [MEMBER]
  ;
    REWRITE_TAC [MEMBER_SNOC] THEN
    REPEAT STRIP_TAC THENL [
      ASM_REWRITE_TAC [MEMBER_SNOC]
    ;
      RES_TAC THEN
      ASSUME_TAC (SPEC "(f:*->bool) x" EXCLUDED_MIDDLE) THEN
      POP_ASSUM DISJ_CASES_TAC THEN
      ASM_REWRITE_TAC [MEMBER_SNOC]
    ]
  ]
);;


let member_projection_list = prove_thm(`member_projection_list`,
  "(!(l:(* list)) (f:(*->bool)) (x:*) .
     (MEMBER x (project_f_lst f l) ==> MEMBER x l)
  )",
  SNOC_INDUCT_TAC THEN
  REPEAT GEN_TAC THEN
  REWRITE_TAC [project_f_lst] THEN
  ASSUME_TAC (SPEC "(f:*->bool) x" EXCLUDED_MIDDLE) THEN
  POP_ASSUM DISJ_CASES_TAC THEN
  ASM_REWRITE_TAC [] THENL [
    REWRITE_TAC [MEMBER_SNOC] THEN
    STRIP_TAC THENL [
      ASM_REWRITE_TAC []
    ;
      RES_TAC THEN
      ASM_REWRITE_TAC []
    ]
  ;
    STRIP_TAC THEN
    IMP_RES_TAC member_projection THEN
    RES_TAC THEN
    ASM_REWRITE_TAC [MEMBER_SNOC]
  ]
);;


let suc_less_eq = prove_thm(`suc_less_eq`,
  "(!m n.
    (m <= n) ==> (m <= SUC n)
  )",
  CONV_TAC ARITH_CONV
);;
```

```
let less_suc_eq_less_eq = prove_thm(`less_suc_eq_less_eq`,
  "(!m n.
    (m <= n) = (m < SUC n)
  )",
  CONV_TAC ARITH_CONV
);;


let length_projection = prove_thm(`length_projection`,
  "(!(l:(* list)) (f:(*->bool)) .
     (LENGTH (project_f_lst f l)) <= (LENGTH l)
  )",
  SNOC_INDUCT_TAC THEN
  REWRITE_TAC [project_f_lst] THENL [
    REWRITE_TAC [LESS_EQ_REFL]
  ;
    REPEAT GEN_TAC THEN
    COND_CASES_TAC THENL [
      ASM_REWRITE_TAC [LENGTH_SNOC; LESS_EQ_MONO]
    ;
      REWRITE_TAC [LENGTH_SNOC] THEN
      mod_ASM_PICK_TAC
        "!f. (LENGTH(project_f_lst f (l:(* list)))) <=
            (LENGTH (l:(* list)))"
        (SPEC "f:(*->bool)") THEN
      IMP_RES_TAC suc_less_eq
    ]
  ]
);;


let project_snoc_true = prove_thm(`project_snoc_true`,
  "(!(l:(* list)) (f:(*->bool)) (x : *) .
     (f x) ==> ((project_f_lst f(SNOC x l)) =
               (SNOC x(project_f_lst f l)))
  )",
  REWRITE_TAC [project_f_lst] THEN
  REPEAT STRIP_TAC THEN
  ASM_REWRITE_TAC []
);;


let project_snoc_false = prove_thm(`project_snoc_false`,
  "(!(l:(* list)) (f:(*->bool)) (x : *) .
     ~(f x) ==> ((project_f_lst f(SNOC x l)) =
                 (project_f_lst f l))
  )",
  REWRITE_TAC [project_f_lst] THEN
  REPEAT STRIP_TAC THEN
  ASM_REWRITE_TAC []
);;



let length_project_snoc_true =
  prove_thm(`length_project_snoc_true`,
  "(!(l:(* list)) (f:(*->bool)) (x : *) .
     (f x) ==> ((LENGTH(project_f_lst f(SNOC x l))) =
               (SUC(LENGTH(project_f_lst f l))))
  )",
  REWRITE_TAC [project_f_lst] THEN
  REPEAT STRIP_TAC THEN
  ASM_REWRITE_TAC [LENGTH_SNOC]
);;


let length_project_snoc_false =
  prove_thm(`length_project_snoc_false`,
  "(!(l:(* list)) (f:(*->bool)) (x : *) .
     ~(f x) ==> ((LENGTH(project_f_lst f(SNOC x l))) =
                (LENGTH(project_f_lst f l)))
  )",
  REWRITE_TAC [project_f_lst] THEN
  REPEAT STRIP_TAC THEN
  ASM_REWRITE_TAC []
);;



let length_project_leq_snoc =
  prove_thm(`length_project_leq_snoc`,
  "(!(l:(* list)) (f:(*->bool)) (x : *) .
     (LENGTH(project_f_lst f l)) <=
     (LENGTH(project_f_lst f(SNOC x l)))
  )",
  REPEAT GEN_TAC THEN
  REWRITE_TAC [project_f_lst] THEN
  ASSUME_TAC (SPEC "(f:*->bool) x" EXCLUDED_MIDDLE) THEN
  POP_ASSUM DISJ_CASES_TAC THENL [
    ASM_REWRITE_TAC [LENGTH_SNOC;LESS_EQ_SUC_REFL;]
  ;
    ASM_REWRITE_TAC [LESS_EQ_REFL]
  ]
);;


let last_cons = prove_thm(`last_cons`,
  "(! (lst:* list) (h:*).
     ~(NULL lst) ==> ((LAST (CONS h lst)) = (LAST lst))
  )",
  LIST_INDUCT_TAC THEN
  REWRITE_TAC [NULL] THEN
  REWRITE_TAC [LAST_DEF] THEN
  REWRITE_TAC [LENGTH;PRE] THEN
  REWRITE_TAC [num_CONV "1";SEG]
);;


let el_length_last = prove_thm(`el_length_last`,
  "(! (lst:* list) (h:*) .
      ((EL (LENGTH lst) (CONS h lst)) = (LAST (CONS h lst)))
  )",
  LIST_INDUCT_TAC THENL
  [
    REWRITE_TAC [LENGTH;EL;LAST_DEF;SEG;PRE;num_CONV "1"]
  ;
    REPEAT GEN_TAC THEN
    ASSUME_TAC
     (SPECL ["h:*";"lst:(* list)"]
       (CONJUNCT2 NULL_DEF)
     ) THEN
    REWRITE_ASM_PICK_TAC
      "NULL(CONS (h:*) lst) = F"
      [SYM_RULE NOT_DEF]
      [] THEN
    IMP_RES_TAC (SPEC "(CONS (h:*) lst)" last_cons) THEN
    ASM_REWRITE_TAC [LENGTH;EL;TL]
  ]
);;
```

```
let length_snoc_tl = prove_thm(`length_snoc_tl`,
  "(! (lst:* list) (x:*) .
     ~(NULL lst) ==>
      ((LENGTH (SNOC x(TL lst))) =
       (LENGTH lst)
      )
  )",
  REPEAT STRIP_TAC THEN
  REWRITE_TAC [LENGTH_SNOC] THEN
  IMP_RES_TAC (SPEC "lst:* list" CONS) THEN
  sym_ASM_PICK_TAC
    "CONS(HD lst)(TL lst) = (lst:* list)"
  THEN
  ONCE_ASM_REWRITE_TAC [] THEN
  REWRITE_TAC [LENGTH;TL]
);;



let length_snoc_snoc = prove_thm(`length_snoc_snoc`,
  "(! (lst:* list) (x x':*) .
     ~(NULL lst) ==>
      ((LENGTH (SNOC x'(SNOC x(TL lst)))) =
      (LENGTH (SNOC x' lst))
      )
  )",
  REPEAT STRIP_TAC THEN
  REWRITE_TAC [LENGTH_SNOC] THEN
  IMP_RES_TAC (SPEC "lst:* list" CONS) THEN
  sym_ASM_PICK_TAC
    "CONS(HD lst)(TL lst) = (lst:* list)"
  THEN
  ASSUM_LIST(\asl.
   GEN_REWRITE_TAC
     (RAND_CONV o ONCE_DEPTH_CONV) [] [(el 1 asl)])
  THEN
  REWRITE_TAC [LENGTH]
);;


let el_length_snoc = prove_thm(`el_length_snoc`,
  "(! (lst:* list) (x:*) .
   EL (LENGTH lst) (SNOC x lst) = x
)",
REPEAT GEN_TAC THEN
ASSUME_TAC (SPECL ["x:*";"lst:* list"] SNOC_NOT_NULL) THEN
IMP_RES_TAC CONS THEN
sym_ASM_PICK_TAC
  "CONS(HD(SNOC x lst))(TL(SNOC x lst)) = SNOC (x:*) lst"
THEN
ONCE_ASM_REWRITE_TAC [] THEN
ASSUME_TAC (SPEC "LENGTH (lst:* list)" LESS_0_CASES) THEN
POP_ASSUM DISJ_CASES_TAC THENL [
  sym_ASM_PICK_TAC
    "0 = LENGTH (lst:* list)" THEN
  IMP_RES_TAC LENGTH_NIL THEN
  ASM_REWRITE_PICK_TAC
    ["(lst:* list) = []"]
    [LENGTH_NIL] THEN
  REWRITE_TAC [LENGTH;EL;HD;SNOC]
;
  IMP_RES_TAC LENGTH_NOT_NULL THEN
  IMP_RES_TAC length_snoc_tl THEN
  sym_ASM_PICK_TAC
    "!x. LENGTH(SNOC (x:*)(TL lst)) = LENGTH lst" THEN
  mod_ASM_PICK_TAC
   "!(x:*). LENGTH lst = LENGTH(SNOC x(TL lst))"
   (SPEC "x:*") THEN
  ASM_REWRITE_PICK_TAC
    ["LENGTH lst = LENGTH(SNOC (x:*)(TL lst))"]
    [] THEN
  KEEP_ASM_TAC
    ["LENGTH lst = LENGTH(SNOC (x:*)(TL lst))";
     "~NULL (lst:* list)";
     "~NULL(SNOC (x:*) lst)";
    ] THEN
  ASM_REWRITE_TAC [TL_SNOC] THEN
  REWRITE_TAC [el_length_last] THEN
  ASSUME_TAC (SPECL ["x:*";"lst:* list"] TL_SNOC) THEN
  REWRITE_ASM_PICK_TAC
    "TL(SNOC (x:*) lst) = (NULL lst => [] | SNOC x(TL lst))"
    []
    ["~NULL (lst:* list)" ] THEN
  sym_ASM_PICK_TAC
    "TL(SNOC (x:*) lst) = SNOC x(TL lst)" THEN
  IMP_RES_TAC CONS THEN
```

```
    ASM_REWRITE_TAC [LAST]
  ]
);;



let project1_def = prove_thm(`project1_def`,
  "(!(l:(* list)) (f:(*->bool)) (eli : *) (i1 : num) .
     ((f eli) /\
      (i1 < LENGTH l) /\
      (EL i1 l = eli)
     ) ==>
     (? i2 .
       (i2 < LENGTH (project_f_lst f l)) /\
       (EL i2 (project_f_lst f l) = eli) /\
       (i2 <= i1)
     )
   )",
  SNOC_INDUCT_TAC THEN
  REPEAT STRIP_TAC THEN
  REWRITE_TAC [project_f_lst] THENL [
    EXISTS_TAC "i1:num" THEN
    ASM_REWRITE_TAC [LESS_EQ_REFL]
  ;
    REWRITE_ASM_PICK_TAC
      "i1 < (LENGTH(SNOC (x:*) l))"
      [LENGTH_SNOC;SYM_RULE less_suc_eq_less_eq;LESS_OR_EQ]
      []
    THEN
    POP_ASSUM DISJ_CASES_TAC THENL [
      ASSUM_LIST
        (\asl. ASSUME_TAC
          (MP (SPECL [
           "i1:num";
           "l:(* list)";
           ] EL_SNOC
          )
          (el 1 asl)
        )
      ) THEN
      REWRITE_ASM_PICK_TAC
        "EL i1(SNOC (x:*) l) = eli"
        []
        ["!(x:*). EL i1(SNOC x l) = EL i1 l"]
  THEN
  mod_ASM_PICK_TAC
   "!f eli i1.
     f eli /\ i1 < (LENGTH (l:(* list))) /\ (EL i1 l = eli)
      ==>
      (?i2.
        i2 < (LENGTH(project_f_lst f l)) /\
        (EL i2 (project_f_lst f l) = eli) /\
        i2 <= i1)"
    (SPECL ["f:(*->bool)";"eli:*";"i1:num"])
  THEN
  RES_TAC THEN
  COND_CASES_TAC THEN
  REWRITE_TAC [LENGTH_SNOC] THEN
  EXISTS_TAC "i2:num" THEN
  IMP_RES_TAC
   (SPECL ["i2:num";"(LENGTH(project_f_lst f (l:* list)))"]
     LESS_SUC)
  THEN
  ASM_REWRITE_TAC [] THEN
  ASSUM_LIST
   (\asl. ASSUME_TAC
      (MP (SPECL [
       "i2:num";
       "(project_f_lst f l:(* list))";
       ] EL_SNOC
      )
      (el 17 asl)
      )
  )
  THEN
  ASM_REWRITE_TAC []
;
  EXISTS_TAC "(LENGTH (project_f_lst f (l:* list)))" THEN
  ASSUME_TAC (SPECL ["l:* list";"x:*"] el_length_snoc) THEN
  TRASH_ASSUM
    "!(f:*->bool) eli i1.
    f eli /\ i1 < (LENGTH l) /\ (EL i1 l = eli) ==>
    (?i2.
      i2 < (LENGTH(project_f_lst f l)) /\
      (EL i2(project_f_lst f l) = eli) /\
      i2 <= i1)" THEN
  REWRITE_ASM_PICK_TAC
    "EL i1(SNOC (x:*) l) = eli"
```

```
          []
          ["i1 = LENGTH (l:* list)"] THEN
        REWRITE_ASM_PICK_TAC
          "EL(LENGTH l)(SNOC x l) = (x:*)"
          []
          ["EL(LENGTH l)(SNOC x l) = eli:*"] THEN
        REWRITE_ASM_PICK_TAC
          "(f:*->bool) eli"
          []
          ["eli = x:*" ] THEN
        ASM_REWRITE_TAC [] THEN
        REWRITE_TAC
          [LENGTH_SNOC;length_projection;
          LESS_SUC_REFL;el_length_snoc]
      ]
    ]
  );;


  let el_snoc_same = prove_thm(`el_snoc_same`,
    "(!(l:(* list)) (eli x : *) (i : num) .
       ((i < LENGTH l) /\ (EL i(SNOC x l) = eli)) ==>
       (EL i l = eli)
     )",
    REPEAT STRIP_TAC THEN
    IMP_RES_TAC EL_SNOC THEN
    sym_ASM_PICK_TAC
      "EL i(SNOC (x:*) l) = eli" THEN
    ASM_REWRITE_TAC []
  );;



  let project_order = prove_thm(`project_order`,
    "(!(l:(* list)) (f:(*->bool)) (el1 el2 : *) (i1 j1 : num) .
       ((f eli) /\ (f elj) /\
        (i1 < LENGTH l) /\
        (j1 < LENGTH l) /\
        (EL i1 l = eli) /\
        (EL j1 l = elj) /\
        (i1 <= j1)
       ) ==>
       (? i2 j2 .
      ((i2 < LENGTH (project_f_lst f l)) /\
       (j2 < LENGTH (project_f_lst f l)) /\
       (EL i2 (project_f_lst f l) = eli) /\
       (EL j2 (project_f_lst f l) = elj) /\
       (i2 <= j2)
      )
    )
  )",
SNOC_INDUCT_TAC THEN
REWRITE_TAC [project_f_lst] THENL [
  REWRITE_TAC [LENGTH;NOT_LESS_0]
;
  REPEAT STRIP_TAC THEN
  COND_CASES_TAC THENL [
    REWRITE_ASM_PICK_TAC
      "j1 < (LENGTH(SNOC (x:*) l))"
      [LENGTH_SNOC;SYM_RULE less_suc_eq_less_eq;LESS_OR_EQ]
      []
     THEN
    POP_ASSUM DISJ_CASES_TAC THENL [
      ASSUM_LIST
          (\asl. ASSUME_TAC
            (SYM_RULE
            (MP (SPECL ["i1:num";"j1:num";"LENGTH(l:* list)"]
             LESS_EQ_LESS_TRANS
            )
            (CONJ (el 3 asl) (el 1 asl))
            )
          )) THEN
      ASSUM_LIST
          (\asl. ASSUME_TAC
            (MP (SPECL ["l:* list";"elj:*";"x:*";"j1:num"]
             el_snoc_same
            )
            (CONJ (el 2 asl) (el 5 asl))
            )
          ) THEN
      ASSUM_LIST
          (\asl. ASSUME_TAC
            (MP (SPECL ["l:* list";"eli:*";"x:*";"i1:num"]
             el_snoc_same
            )
            (CONJ (el 2 asl) (el 7 asl))
            )
    ) THEN
ASSUM_LIST
    (\asl. ASSUME_TAC
      (MP
    (SPECL
       ["f:*->bool";"eli:*";"elj:*";"i1:num";"j1:num"]
       (el 12 asl)
    )
    (LIST_CONJ
      [(el 11 asl);(el 10 asl);(el 3 asl);(el 4 asl);
       (el 1 asl);(el 2 asl);(el 6 asl)]
    )
      )
    ) THEN
POP_ASSUM (X_CHOOSE_TAC "i2:num") THEN
POP_ASSUM (X_CHOOSE_TAC "j2:num") THEN
MAP_EVERY EXISTS_TAC ["i2:num";"j2:num"] THEN
UNDISCH_TAC
  "i2 < (LENGTH(project_f_lst f (l:* list))) /\
   j2 < (LENGTH(project_f_lst f l)) /\
   (EL i2(project_f_lst f l) = eli) /\
   (EL j2(project_f_lst f l) = elj) /\
   i2 <= j2"
THEN
STRIP_TAC THEN
ASSUM_LIST
    (\asl. ASSUME_TAC
      (MP
       (SPECL
     ["i2:num";"LENGTH(project_f_lst f (l:* list))"]
    (LESS_SUC)
       )
       (el 5 asl)
      )
    ) THEN
ASSUM_LIST
    (\asl. ASSUME_TAC
      (MP
       (SPECL
     ["j2:num";"LENGTH(project_f_lst f (l:* list))"]
    (LESS_SUC)
       )
       (el 5 asl)
      )
      ) THEN
    ASSUM_LIST
       (\asl. ASSUME_TAC
          (MP (SPECL [
           "i2:num";
           "(project_f_lst f l:(* list))";
           ] EL_SNOC
          )
          (el 7 asl)
          )
      ) THEN
    ASSUM_LIST
       (\asl. ASSUME_TAC
          (MP (SPECL [
           "j2:num";
           "(project_f_lst f l:(* list))";
           ] EL_SNOC
          )
          (el 7 asl)
          )
      ) THEN
    ASM_REWRITE_TAC [LENGTH_SNOC]
  ;
    TRASH_ASSUM
    "!(f:*->bool) (el1 el2:*) i1 j1.
        f eli /\
        f elj /\
        i1 < (LENGTH l) /\
        j1 < (LENGTH l) /\
        (EL i1 l = eli) /\
        (EL j1 l = elj) /\
        i1 <= j1 ==>
        (?i2 j2.
          i2 < (LENGTH(project_f_lst f l)) /\
          j2 < (LENGTH(project_f_lst f l)) /\
          (EL i2(project_f_lst f l) = eli) /\
          (EL j2(project_f_lst f l) = elj) /\
          i2 <= j2)"
      THEN
    ASSUM_LIST
       (\asl. ASSUME_TAC
          (MP
           (SPECL ["(SNOC (x:*) l)";
                "f:*->bool";
                "eli:*";
                "i1:num"]
           project1_def
            )
            (LIST_CONJ [(el 8 asl);(el 6 asl);(el 5 asl)])
           )
      ) THEN
   POP_ASSUM (X_CHOOSE_TAC "i2:num") THEN
   UNDISCH_TAC
   "i2 < (LENGTH(project_f_lst f(SNOC (x:*) l))) /\
      (EL i2(project_f_lst f(SNOC x l)) = eli) /\
      i2 <= i1"
    THEN
   STRIP_TAC THEN
   MAP_EVERY EXISTS_TAC
     ["i2:num";"LENGTH(project_f_lst f (l:* list))"] THEN
   ASSUM_LIST
       (\asl. ASSUME_TAC
         (SYM_RULE
         (MP (SPECL ["l:* list";"f:*->bool";"x:*"]
           project_snoc_true
         )
         (el 5 asl)
         )
       )) THEN
   ASM_REWRITE_TAC [] THEN
   sym_ASM_PICK_TAC
    "SNOC (x:*)(project_f_lst f l) = project_f_lst f(SNOC x l)"
    THEN
   REWRITE_ASM_PICK_TAC
    "i2 < (LENGTH(project_f_lst f(SNOC (x:*) l)))"
    [LENGTH_SNOC;SYM_RULE less_suc_eq_less_eq]
    ["project_f_lst f(SNOC (x:*) l) = SNOC x(project_f_lst f l)"]
    THEN
   IMP_RES_TAC length_project_snoc_true THEN
   ASM_REWRITE_TAC [LESS_SUC_REFL] THEN
   KEEP_ASM_TAC
    ["j1 = LENGTH (l:* list)";
     "EL j1(SNOC (x:*) l) = elj";
    ] THEN
   REWRITE_ASM_PICK_TAC
     "EL j1(SNOC (x:*) l) = elj"
     [el_length_snoc]
     ["j1 = LENGTH (l:* list)"]
     THEN
    ASM_REWRITE_TAC [el_length_snoc]
  ]
;
  REWRITE_ASM_PICK_TAC
    "j1 < (LENGTH(SNOC (x:*) l))"
    [LENGTH_SNOC;SYM_RULE less_suc_eq_less_eq;LESS_OR_EQ]
    []
    THEN
  POP_ASSUM DISJ_CASES_TAC THENL [
    ASSUM_LIST
        (\asl. ASSUME_TAC
          (SYM_RULE
          (MP (SPECL ["i1:num";"j1:num";"LENGTH(l:* list)"]
           LESS_EQ_LESS_TRANS
          )
          (CONJ (el 3 asl) (el 1 asl))
          )
        )) THEN
    ASSUM_LIST
        (\asl. ASSUME_TAC
          (MP (SPECL ["l:* list";"elj:*";"x:*";"j1:num"]
           el_snoc_same
          )
          (CONJ (el 2 asl) (el 5 asl))
          )
        ) THEN
    ASSUM_LIST
        (\asl. ASSUME_TAC
          (MP (SPECL ["l:* list";"eli:*";"x:*";"i1:num"]
           el_snoc_same
          )
          (CONJ (el 2 asl) (el 7 asl))
          )
        ) THEN
    ASSUM_LIST
        (\asl. ASSUME_TAC
          (MP
        (SPECL
           ["f:*->bool";"eli:*";"elj:*";"i1:num";"j1:num"]
           (el 12 asl)
        )
        (LIST_CONJ
          [(el 11 asl);(el 10 asl);(el 3 asl);(el 4 asl);
        (el 1 asl);(el 2 asl);(el 6 asl)]
     )
       )
     ) THEN
POP_ASSUM (X_CHOOSE_TAC "i2:num") THEN
POP_ASSUM (X_CHOOSE_TAC "j2:num") THEN
MAP_EVERY EXISTS_TAC ["i2:num";"j2:num"] THEN
UNDISCH_TAC
  "i2 < (LENGTH(project_f_lst f (l:* list))) /\
   j2 < (LENGTH(project_f_lst f l)) /\
   (EL i2(project_f_lst f l) = eli) /\
   (EL j2(project_f_lst f l) = elj) /\
   i2 <= j2"
THEN
STRIP_TAC THEN
ASSUM_LIST
     (\asl. ASSUME_TAC
       (MP
        (SPECL
      ["i2:num";"LENGTH(project_f_lst f (l:* list))"]
     (LESS_SUC)
        )
        (el 5 asl)
       )
     ) THEN
ASSUM_LIST
     (\asl. ASSUME_TAC
       (MP
        (SPECL
      ["j2:num";"LENGTH(project_f_lst f (l:* list))"]
     (LESS_SUC)
        )
        (el 5 asl)
       )
     ) THEN
ASSUM_LIST
   (\asl. ASSUME_TAC
       (MP (SPECL [
        "i2:num";
        "(project_f_lst f l:(* list))";
        ] EL_SNOC
       )
       (el 7 asl)
       )
```

```
          ) THEN
         ASSUM_LIST
            (\asl. ASSUME_TAC
               (MP (SPECL [
                "j2:num";
                "(project_f_lst f l:(* list))";
                ] EL_SNOC
               )
               (el 7 asl)
               )
          ) THEN
         ASM_REWRITE_TAC [LENGTH_SNOC]
       ;
         REWRITE_ASM_PICK_TAC
           "EL j1(SNOC (x:*) l) = elj"
           [el_length_snoc]
           ["j1 = LENGTH (l:* list)"]
          THEN
         REWRITE_ASM_PICK_TAC
           "~(f:*->bool) x"
           []
           ["(x:*) = elj"]
          THEN
         KEEP_ASM_TAC
           ["~(f:*->bool) elj";
            "(f:*->bool) elj"
           ]
          THEN
         RES_TAC
       ]
     ]
   ]
);;


close_theory ();;
```

## A.11 mapdefs.ml

```
%-------------------------------------
* File:    mapdefs.ml
* Version: 0.0
* Date:    11/18/96

961011. Fixed map_up_FGs to check if fromF_FG
        instead of just fromF to simplify
        map up of FG operations.
961014. Changed mk_fg_msg_list and mk_g_msg_list
        to use get_mbx_unread_msgs.
961023. Added transition labels.
*-------------------------------------%

% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Definitions for mapping of composition of F and FG servers to
OS-sys.

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

loadt `aux/init.ml`;; loadt `aux/letconv.ml`;; new_theory_safe
`mapdefs`;; load_library `sets`;; load_parent `silobasic`;;
load_parent `silomorelists`;; load_parent `silolists`;;
load_parent `siloprojection`;; load_parent `F-FG`;; load_parent
`OS-env`;; load_parent `OS-sys`;; load_parent `silomappings`;;



% ----------------------------------------------------------------
* fromF

True if the sender id of the input message is F_ID.
-------------------------------------------------------------- %

let fromF = new_definition(`fromF`,
  "(! msg : msg_def .
     fromF msg = (get_msg_sndr msg = F_ID)
  )"
);;


% ----------------------------------------------------------------
* fromFG

True if the sender id of the input message is FG_ID.
-------------------------------------------------------------- %
```

```
let fromFG = new_definition(`fromFG`,
  "(! msg : msg_def .
     fromFG msg = (get_msg_sndr msg = FG_ID)
  )"
);;


% ------------------------------------------------------------------
* fromF_FG

------------------------------------------------------------------ %

let fromF_FG = new_definition(`fromF_FG`,
  "(! msg : msg_def .
     fromF_FG msg =
      ((fromF msg) \/ (fromFG msg))
  )"
);;



% ------------------------------------------------------------------
* filtmbxF_FG

Given a mailbox, returns the same mailbox minus any messages sent
by the F or FG servers. The presence and relative order in the
mailbox of all other messages is unaffected. The composition of ~
and fromF_FG insures that the filter passes only messages that
were not sent by the servers.
------------------------------------------------------------------ %

let filtmbxF_FG = new_definition(`filtmbxF_FG`,
  "(! mbx:mbox_def .
     filtmbxF_FG mbx =
     let mnext = get_mbx_next mbx
     and msgs = get_mbx_msgs mbx
     in
     let newmnext =
       (LENGTH (project_f_lst ($~ o fromF_FG)
                              (SUBSEQ mnext msgs)))
     in (
       cons_mbx newmnext (project_f_lst ($~ o fromF_FG) msgs)
     )
  )"
```

```
);;


% ------------------------------------------------------------------
* projectF

Given a mailbox, returns only the messages sent by the F server,
in their original order.
------------------------------------------------------------------ %

let projectF = new_definition(`projectF`,
  "(! mbx:mbox_def .
     projectF mbx =
     let mnext = get_mbx_next mbx
     and msgs = get_mbx_msgs mbx
     in
     let newmnext =
       (LENGTH (project_f_lst (fromF)
                               (SUBSEQ mnext msgs)))
     in (
       cons_mbx newmnext (project_f_lst (fromF) msgs)
     )
  )"
);;


% ------------------------------------------------------------------
* projectFG

Given a mailbox, returns only the messages sent by the FG server,
in their original order.
------------------------------------------------------------------ %

let projectFG = new_definition(`projectFG`,
  "(! mbx:mbox_def .
     projectFG mbx =
     let mnext = get_mbx_next mbx
     and msgs = get_mbx_msgs mbx
     in
     let newmnext =
       (LENGTH (project_f_lst (fromFG)
                               (SUBSEQ mnext msgs)))
     in (
      cons_mbx newmnext (project_f_lst (fromFG) msgs)
    )
  )"
);;


% --------------------------------------------------------------
* map_up_mbxs

Given the mailboxes of a system state, returns all mailboxes
unmodified, except the F and FG mailboxes, from which all the
messages from the F and FG servers have been filtered.
-------------------------------------------------------------- %

let map_up_mbxs = new_definition(`map_up_mbxs`,
  "(! mbxs:mboxes .
     map_up_mbxs mbxs =
     let oldFmbx = (get_mbxs_mbx mbxs F_ID)
     and oldFGmbx = (get_mbxs_mbx mbxs FG_ID)
     in
     let newFmbx = (filtmbxF_FG oldFmbx)
     and newFGmbx = (filtmbxF_FG oldFGmbx)
     in (
       put_mbxs_mbx
         (put_mbxs_mbx mbxs F_ID newFmbx)
         FG_ID
         newFGmbx
     )
  )"
);;


% --------------------------------------------------------------
* map_up_Fs

Given a system state at the server level, returns the internal
state queue for the F service call.
-------------------------------------------------------------- %

let map_up_Fs = new_definition(`map_up_Fs`,
  "(!fi:F_internal_state.
     map_up_Fs fi =
%     let fi = get_trace_Fs ss
```

```
      in  %
      let fi_flag = Fs_get_flag fi
      and fi_buf = get_Fs_buf fi
      in (
%If empty buffer then return empty queue%
       (~fi_flag) => ([]:msg_def list) |
%else filter out messages from the FG server.%
       ((fromF_FG fi_buf)) =>
          ([]:msg_def list) |
       ([fi_buf])
      )
  )"
);;


% ------------------------------------------------------------------
* get_mbx_unread_msgs

Returns the unread messages in a mailbox, or an empty list if
there are no unread messages.
------------------------------------------------------------------ %

let get_mbx_unread_msgs = new_definition (`get_mbx_unread_msgs`,
  "(! (mbx : mbox_def) .
  get_mbx_unread_msgs mbx =
  let msgs = (get_mbx_msgs mbx)
  and nxt = (get_mbx_next mbx)
  in
    (mbx_is_unread_msg mbx) => (newBUTFIRSTN nxt msgs) |
                               []:(msg_def)list
  )"
);;


% ------------------------------------------------------------------
* mk_g_msg_list

Given a system state at the server level, returns a list of
messages from the F mailbox and internal state that corresponds to
messages in the FG server internal queue. These messages were
already processed by the FG server and their data values have
already had the G function applied to them.
------------------------------------------------------------------ %
```

```
let mk_g_msg_list = new_definition(`mk_g_msg_list`,
  "(! ss: trace_state.
    mk_g_msg_list ss =
     let fi = get_trace_Fs ss
     and fmbx = (get_mbxs_mbx(get_trace_mbxs ss)F_ID)
     in
     let fi_flag = Fs_get_flag fi
     and fi_buf = get_Fs_buf fi
     and fmbxmsgs = (get_mbx_unread_msgs fmbx)
     in
     let fmsgs = (project_f_lst fromFG fmbxmsgs)
     and fbuf = (((~fi_flag) \/ (~fromFG fi_buf)) =>
                     ([]:msg_def list) |
                     [fi_buf]
                )
     in (
      APPEND fbuf fmsgs
    )
  )"
);;



% ---------------------------------------------------------------
* mk_fg_msg_list

Given a system state at the server level, returns a list of
messages from the FG mailbox and internal state that corresponds
to messages in the FG server internal queue. These messages were
already processed by the F server and have F(G(x)) as their data
values.
--------------------------------------------------------------- %

let mk_fg_msg_list = new_definition(`mk_fg_msg_list`,
  "(! ss: trace_state.
    mk_fg_msg_list ss =
     let fgi = get_trace_FGs ss
     and fgmbx = (get_mbxs_mbx(get_trace_mbxs ss)FG_ID)
     in
     let fgi_flag = FGs_get_flag fgi
     and fgi_buf = get_FGs_buf fgi
     and fgmbxmsgs = (get_mbx_unread_msgs fgmbx)
     in
     let fgmsgs = (project_f_lst fromF fgmbxmsgs)
     and fgbuf = (((~fgi_flag) \/ (~fromF fgi_buf)) =>
                    ([]:msg_def list) |
                    [fgi_buf]
                 )
     in (
      APPEND fgbuf fgmsgs
    )
  )"
);;



% ------------------------------------------------------------------
* mk_par_msg_list

Given a system state at the server level, returns a list of
messages from the os mailboxes and internal state that corresponds
to the messages in the FG server internal queue. These messages
have either G(x) or F(G(x)) as their data values.
------------------------------------------------------------------ %

let mk_par_msg_list = new_definition(`mk_par_msg_list`,
  "(! ss: trace_state.
     mk_par_msg_list ss =
     let glst = (mk_g_msg_list ss)
     and fglst =(mk_fg_msg_list ss)
     in (APPEND fglst glst)
  )"
);;



% ------------------------------------------------------------------
* map_up_FGs

Given a system state, returns the internal state queue for the FG
service call.
------------------------------------------------------------------ %

let map_up_FGs = new_definition(`map_up_FGs`,
  "(! fgi: FG_internal_state.
     map_up_FGs fgi =
%     let fgi = get_trace_FGs ss
```

```
      in %
      let fgi_flag = FGs_get_flag fgi
      and fgi_buf = get_FGs_buf fgi
      and fgi_q = FGs_get_queue fgi
      in (
        ((~fgi_flag) \/ (fromF_FG fgi_buf)) =>
                 fgi_q |
                (SNOC fgi_buf fgi_q)
      )
   )"
);;


% ---------------------------------------------------------------
* map_up_state

Maps the system state for the F and FG server composition to the
system state of the OS.
--------------------------------------------------------------- %

let map_up_state = new_definition(`map_up_state`,
  "(!(lowss:trace_state) .
     map_up_state lowss =
     let mbxs = (get_trace_mbxs lowss)
     in
     let newmbxs = map_up_mbxs mbxs
     and newfs = map_up_Fs(get_trace_Fs lowss)
     and newfgs = map_up_FGs(get_trace_FGs lowss)
     in (cons_OStrace_state newmbxs newfs newfgs)
  )"
);;



% ---------------------------------------------------------------
* map_up_element

Maps a trace element from a composed system trace to a trace
element of the OS.
--------------------------------------------------------------- %

let map_up_element = new_definition(`map_up_element`,
  "(!(lowtel:trace_element) .
     map_up_element lowtel =
     let ltagent = get_trace_agent lowtel
     and ltlabel = get_trace_tlabel lowtel
     and ltstate = get_trace_state lowtel
     in
       cons_OStrace_el ltagent ltlabel (map_up_state ltstate)
  )"
);;



% ---------------------------------------------------------------
* map_up_trace

Maps a trace in the composed system to a trace in the complete OS
specification.
--------------------------------------------------------------- %

let map_up_trace = new_definition(`map_up_trace`,
  "(!(lowtrace:trace_def) .
     map_up_trace lowtrace = (map_up_element o lowtrace)
  )"
);;


close_theory();;
```

# Appendix B

# Example Template Instantiations

This appendix contains two instantiations of the example template from chapter 7, as described in section 7.10. The functions are defined as $F\_func$, $G\_func$, and $FG\_func$, and the proof that $(F\_func(G\_func(x)) = FG\_func(x))$ is defined as $proveF\_FG$.

The only difference between the two instantiations of the template are the definitions of the specific functions $F\_func$, $G\_func$, and $FG\_func$, and the proof $proveF\_FG$. Otherwise, the proofs are identical and can be carried out almost automatically.

## B.1 First Instantiation

```
%-------------------------------------
* File:    instance1.ml
*-------------------------------------%


loadt `aux/init.ml`;;
loadt `aux/letconv.ml`;;
new_theory_safe `instance1`;;
load_library `sets`;;
```

```
load_parent 'silobasic';;
load_parent 'F-FG';;
load_parent 'OS-env';;
load_parent 'OS-sys';;
load_library 'arith';;
load_library 'more_arithmetic';;
load_parent 'composeF-FG';;
load_parent 'safety';;
load_parent 'mapFprogress';;
load_parent 'mapFGreadsprog';;
load_parent 'mapFGrespondsprog';;


% -----------------------------------------------------------------
* F_func

The definition of the system call F.

This system call accepts a value 'v' in its request message and
returns a value F(v) = v + 1 in the response.

----------------------------------------------------------------- %

let F_func = new_definition ('F_func', "
  (! (mdata : msg_data) .
  F_func mdata =
    (mdata+1)
  )
");;

% -----------------------------------------------------------------
* G_func

The FG system call accepts a value 'v' in its request message and
passes the value G(v) to the F server.

For demonstration purposes, we specify G(v) as (v+2).
When combined with F(v) = (v+1), the response of the FG server will
be (v+3).
----------------------------------------------------------------- %

let G_func = new_definition ('G_func', "
  (! (mdata : msg_data) .
  G_func mdata =
    (mdata+2)
  )
");;

% -----------------------------------------------------------------
* FG_func

The OS system call FG accepts a value 'v' in its request message and
returns the value FG(v) in the response.

For demonstration purposes, we specify FG(v) as (v+3).
In the proof we show that the composition of the F and FG servers
implements this function.
----------------------------------------------------------------- %

let FG_func = new_definition ('FG_func', "
  (! (mdata : msg_data) .
  FG_func mdata =
    (mdata+3)
  )
");;


% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
The things that we must prove in order to show that the
composition is correct:

1)Composition step.
2)Map up safety properties (initial state and transitions),
  and show data refinement of external state preserves environment
  transitions.
3)Map up liveness properties.
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

let composition_step = new_definition('composition_step',
  "(!trace. composition_step trace =
    ((OS_env_init trace) /\ (OS_env_safety trace) /\
    (F_init trace) /\ (F_safety trace F_func) /\
    (FG_init trace) /\ (FG_safety trace G_func)
    ==>
    (F_env_init trace) /\ (F_env_safety trace) /\
    (FG_env_init trace) /\ (FG_env_safety trace))
  )"
);;
```

```
let map_safety_step = new_definition(`map_safety_step`,
  "(!trace. map_safety_step trace =
    OS_env_init trace /\
    OS_env_safety trace /\
    F_init trace /\
    FG_init trace /\
    F_safety trace F_func /\
    FG_safety trace G_func ==>
     OS_init (map_up_trace trace) /\
     OS_safety (map_up_trace trace) F_func FG_func /\
     (OS_env_init (map_up_trace_mbxs trace) /\
      OS_env_safety (map_up_trace_mbxs trace))
  )"
);;

let map_progress_step = new_definition(`map_progress_step`,
  "(!trace. map_progress_step trace =
     OS_env_init trace /\
     OS_env_safety trace /\
     F_init trace /\
     FG_init trace /\
     F_safety trace F_func /\
     FG_safety trace G_func /\
     F_reads_progress trace /\
     F_responds_progress trace F_func /\
     FG_reads_progress trace /\
     FG_responds_progress trace G_func ==>
      OS_reads_F_progress (map_up_trace trace) /\
      OS_responds_F_progress (map_up_trace trace) F_func /\
      OS_reads_FG_progress (map_up_trace trace) /\
      OS_responds_FG_progress (map_up_trace trace) FG_func
  )"
);;


% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
The assumption that must be true for the composition proof
to succeed.  Viz., F o G = FG
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% ----------------------------------------------------------------
* proveF_FG
```

```
---------------------------------------------------------------- %

let proveF_FG = prove_thm(`proveF_FG`,
  "(!x. (F_func(G_func x) = (FG_func x))
  )",
  GEN_TAC THEN
  REWRITE_TAC [F_func;G_func;FG_func] THEN
  CONV_TAC ARITH_CONV
);;


let prove_instance1 = prove_thm(`prove_instance1`,
  "(!trace.
     composition_step trace /\
     map_safety_step trace /\
     map_progress_step trace
  )",
  GEN_TAC THEN
  CONJ_TAC THENL [
    REWRITE_TAC [composition_step] THEN
    STRIP_TAC THEN
    MP_TAC (SPECL ["trace:trace_def";"F_func:num->num";
                   "G_func:num->num"]
      composeF_FG)
     THEN
    ASM_REWRITE_TAC []
  ;
    CONJ_TAC THENL [
      REWRITE_TAC [map_safety_step] THEN
      STRIP_TAC THEN
      MP_TAC (SPECL ["trace:trace_def"]
        map_up_init)
       THEN
      MP_TAC (SPECL ["trace:trace_def";"F_func:num->num";
                     "G_func:num->num";"FG_func:num->num"]
        map_up_transitions)
       THEN
      MP_TAC (SPECL ["trace:trace_def";"F_func:num->num";
                     "G_func:num->num"]
        map_up_env)
       THEN
      ASM_REWRITE_TAC [assumption;proveF_FG] THEN
      REPEAT STRIP_TAC THEN
      ASM_REWRITE_TAC []
    ;
      REWRITE_TAC [map_progress_step] THEN
      STRIP_TAC THEN
      MP_TAC (SPECL ["trace:trace_def";"F_func:num->num";
                     "G_func:num->num"]
        map_F_responds_progress)
       THEN
      MP_TAC (SPECL ["trace:trace_def";"F_func:num->num";
                     "G_func:num->num"]
        map_F_reads_progress)
       THEN
      MP_TAC (SPECL ["trace:trace_def";"F_func:num->num";
                     "G_func:num->num"]
        map_FG_reads_progress)
       THEN
      MP_TAC (SPECL ["trace:trace_def";"F_func:num->num";
                     "G_func:num->num";"FG_func:num->num"]
        map_FG_responds_progress)
       THEN
      ASM_REWRITE_TAC [assumption;proveF_FG] THEN
      REPEAT STRIP_TAC THEN
      ASM_REWRITE_TAC []
    ]
  ]
);;


close_theory();;
```

## B.2 Second Instantiation

```
%---------------------------------------
* File:    instance2.ml
*---------------------------------------%


loadt `aux/init.ml`;;
loadt `aux/letconv.ml`;;
new_theory_safe `instance1`;;
```

```
load_library `sets`;;
load_parent `silobasic`;;
load_parent `F-FG`;;
load_parent `OS-env`;;
load_parent `OS-sys`;;
load_library `arith`;;
load_library `more_arithmetic`;;
load_parent `composeF-FG`;;
load_parent `safety`;;
load_parent `mapFprogress`;;
load_parent `mapFGreadsprog`;;
load_parent `mapFGrespondsprog`;;


% ----------------------------------------------------------------
* F_func

The definition of the system call F.

This system call accepts a value 'v' in its request message and
returns a value F(v) = v * 2 in the response.

---------------------------------------------------------------- %

let F_func = new_definition (`F_func`, "
  (! (mdata : msg_data) .
  F_func mdata =
    (mdata * 2)
  )
");;

% ----------------------------------------------------------------
* G_func

The FG system call accepts a value 'v' in its request message and
passes the value G(v) to the F server.

For demonstration purposes, we specify G(v) as (v+2).
---------------------------------------------------------------- %

let G_func = new_definition (`G_func`, "
  (! (mdata : msg_data) .
  G_func mdata =
    (mdata+2)
  )
");;

% ----------------------------------------------------------------
* FG_func

The OS system call FG accepts a value 'v' in its request message and
returns the value FG(v) in the response.

For demonstration purposes, we specify FG(v) as (2v+4).
In the proof we show that the composition of the F and FG servers
implements this function.
---------------------------------------------------------------- %

let FG_func = new_definition ('FG_func', "
  (! (mdata : msg_data) .
  FG_func mdata =
    ((2 * mdata) + 4)
  )
");;


% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
The things that we must prove in order to show that the
composition is correct:

1)Composition step.
2)Map up safety properties (initial state and transitions),
  and show data refinement of external state preserves environment
  transitions.
3)Map up liveness properties.
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

let composition_step = new_definition('composition_step',
  "(!trace. composition_step trace =
    ((OS_env_init trace) /\ (OS_env_safety trace) /\
    (F_init trace) /\ (F_safety trace F_func) /\
    (FG_init trace) /\ (FG_safety trace G_func)
    ==>
    (F_env_init trace) /\ (F_env_safety trace) /\
    (FG_env_init trace) /\ (FG_env_safety trace))
  )"
);;
```

```
let map_safety_step = new_definition(`map_safety_step`,
  "(!trace. map_safety_step trace =
    OS_env_init trace /\
    OS_env_safety trace /\
    F_init trace /\
    FG_init trace /\
    F_safety trace F_func /\
    FG_safety trace G_func ==>
     OS_init (map_up_trace trace) /\
     OS_safety (map_up_trace trace) F_func FG_func /\
     (OS_env_init (map_up_trace_mbxs trace) /\
      OS_env_safety (map_up_trace_mbxs trace))
  )"
);;

let map_progress_step = new_definition(`map_progress_step`,
  "(!trace. map_progress_step trace =
     OS_env_init trace /\
     OS_env_safety trace /\
     F_init trace /\
     FG_init trace /\
     F_safety trace F_func /\
     FG_safety trace G_func /\
     F_reads_progress trace /\
     F_responds_progress trace F_func /\
     FG_reads_progress trace /\
     FG_responds_progress trace G_func ==>
      OS_reads_F_progress (map_up_trace trace) /\
      OS_responds_F_progress (map_up_trace trace) F_func /\
      OS_reads_FG_progress (map_up_trace trace) /\
      OS_responds_FG_progress (map_up_trace trace) FG_func
  )"
);;


% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
The assumption that must be true for the composition proof
to succeed.  Viz., F o G = FG
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< %

% ----------------------------------------------------------------
* proveF_FG
```

```
------------------------------------------------------------------ %

let proveF_FG = prove_thm(`proveF_FG`,
  "(!x. (F_func(G_func x) = (FG_func x))
  )",
  GEN_TAC THEN
  REWRITE_TAC [F_func;G_func;FG_func] THEN
  CONV_TAC ARITH_CONV
);;


let prove_instance1 = prove_thm(`prove_instance1`,
  "(!trace.
     composition_step trace /\
     map_safety_step trace /\
     map_progress_step trace
  )",
  GEN_TAC THEN
  CONJ_TAC THENL [
    REWRITE_TAC [composition_step] THEN
    STRIP_TAC THEN
    MP_TAC (SPECL ["trace:trace_def";"F_func:num->num";
                   "G_func:num->num"]
      composeF_FG)
     THEN
    ASM_REWRITE_TAC []
  ;
    CONJ_TAC THENL [
      REWRITE_TAC [map_safety_step] THEN
      STRIP_TAC THEN
      MP_TAC (SPECL ["trace:trace_def"]
        map_up_init)
       THEN
      MP_TAC (SPECL ["trace:trace_def";"F_func:num->num";
                     "G_func:num->num";"FG_func:num->num"]
        map_up_transitions)
       THEN
      MP_TAC (SPECL ["trace:trace_def";"F_func:num->num";
                     "G_func:num->num"]
        map_up_env)
       THEN
      ASM_REWRITE_TAC [assumption;proveF_FG] THEN
      REPEAT STRIP_TAC THEN
      ASM_REWRITE_TAC []
      ;
       REWRITE_TAC [map_progress_step] THEN
       STRIP_TAC THEN
       MP_TAC (SPECL ["trace:trace_def";"F_func:num->num";
                      "G_func:num->num"]
         map_F_responds_progress)
        THEN
       MP_TAC (SPECL ["trace:trace_def";"F_func:num->num";
                      "G_func:num->num"]
         map_F_reads_progress)
        THEN
       MP_TAC (SPECL ["trace:trace_def";"F_func:num->num";
                      "G_func:num->num"]
         map_FG_reads_progress)
        THEN
       MP_TAC (SPECL ["trace:trace_def";"F_func:num->num";
                      "G_func:num->num";"FG_func:num->num"]
         map_FG_responds_progress)
        THEN
       ASM_REWRITE_TAC [assumption;proveF_FG] THEN
       REPEAT STRIP_TAC THEN
       ASM_REWRITE_TAC []
     ]
   ]
);;


close_theory();;
```

# Bibliography

[1] M. Aagaard and M. Leeser. A methodology for reusable hardware proofs. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications: Proceedings of the IFIP TC10/WG10.2 Workshop*, volume A-20 of *IFIP Transactions*, pages 177–196, Leuven, Belgium, September 1992. North-Holland/Elsevier.

[2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[3] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(3):73–132, January 1993.

[4] Gregory. R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley Publishing Company, 1991.

[5] Tej Arora, Tony Leung, Karl Levitt, Thomas Schubert, and Phillip Windley. Report on the UCD microcoded Viper verification project. In *Proceedings of the International Higher-Order-Logic Theorem Proving Workshop*, pages 241–254, Vancouver, B.C., August 1993.

[6] Michael Baentsch, L. Baum, Georg Molter, S. Rothkugel, and P. Sturm. World wide web caching: the application-level view of the internet. *IEEE Communications Magazine*, 35(6):170–178, June 1997.

[7] William R. Bevier. Kit and the short stack. *Journal of Automated Reasoning*, 5:519–530, 1989.

[8] William R. Bevier, Warren A. Hunt, Jr., J. Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5:411–428, 1989.

[9] R. J. Boulton. The HOL arith library. Technical report, University of Cambridge, Computer Laboratory, July 1992.

[10] Robert S. Boyer and J. Strother Moore. *A Computational Logic.* Academic Press, 1979.

[11] Joshua E. Caplan and Mehdi T. Harandi. A logical framework for software proof reuse. In *Proceedings of the ACM-SIGSOFT Symposium on Software Reusability (SSR'95).* ACM-SIGSOFT, 1995.

[12] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: a Foundation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[13] A. Cohn. A proof of correctness of the Viper microprocessor: The first level. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer Academic Publishers, 1988.

[14] Richard A. DeMillo, W. Michael McCracken, R. J. Martin, and John F. Passafiume.

*Software Testing and Evaluation.* Benjamin/Cummings Publishing Company, Inc., Redwood City, California., 1987.

[15] M. J. C. Gordon. Current trends in hardware verification and automated theorem proving. In G. Birtwistle and P. A. Subrahmanyam, editors, *Mechanizing Programming Logics in Higer Order Logic*, pages 387–439. Springer-Verlag, New York, 1989.

[16] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[17] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems.* Springer-Verlag, 1985. NATO ASI Series, Vol. F13.

[18] Mark R. Heckman and Karl N. Levitt. Applying the composition principle to verify a hierarchy of security servers. In *Proceedings of the Thirty-first Hawaii International Conference on System Sciences (HICSS-31)*, January 1998.

[19] Mark R. Heckman, Cui Zhang, Brian R. Becker, David Peticolas, Karl N. Levitt, and Ron A. Olsson. Towards applying the composition principle to verify a microkernel operating system. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, number 1125 in Lecture Notes in Computer Science, pages 235–250, Turku, Finland, August 1996. Springer-Verlag.

[20] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[21] Cliff B. Jones. *Systematic software development using VDM.* Prentice Hall International (UK Ltd., London, 1990.

[22] Bengt Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, March 1994.

[23] J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer Academic Publishers, 1988.

[24] Simon S. Lam and A. Udaya Shankar. A theory of interfaces and modules: I—composition theorem. *IEEE Transactions on Software Engineering*, 20(1):55–71, January 1994.

[25] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[26] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[27] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York, 1992.

[28] T. F. Melham. The HOL sets library. Technical report, University of Cambridge, Computer Laboratory, October 1991. Distributed with the HOL 88.2.0 system.

[29] P. V. Mockapetris. RFC 1034: Domain names — concepts and facilities, November 1987.

[30] J. Strother Moore. System verification. *Journal of Automated Reasoning*, 5:409–410, 1989.

[31] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[32] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[33] O. Sami Saydjari, S. Jeffrey Turner, D. Elmo Peele, John F. Farrell, Peter A. Loscocco, William Kutz, and Gregory L. Bock. Synergy: A distributed, microkernel-based security architecture. Technical report, NSA INFOSEC Research and Technology, November 1993.

[34] Edward Thomas Schubert. *A Methodology for the Formal Verification of Composed Hardware Systems*. PhD thesis, University of California, Davis, 1992.

[35] Ian Sommerville. *Software Engineering - 3rd Edition*. Addison-Wesley Pub. Co., Wokingham, England; Reading, Mass., 1989.

[36] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992.

[37] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation.* Prentice-Hall, Inc., 1997.

[38] Phillip J. Windley. *The Formal Verification of Generic Interpreters.* PhD thesis, University of California, Davis, 1990.

[39] C. Zhang, R. Shaw, M. R. Heckman, G. D. Benson, M. Archer, K. Levitt, and R. A. Olsson. Towards a formal verification of a secure distributed system and its applications. In *Proceedings of the 17th National Computer Security Conference*, Baltimore, October 1994.

[40] C. Zhang, R. Shaw, R. Olsson, K. Levitt, M. Archer, M. Heckman, and G. Benson. Mechanizing a programming logic for the concurrent programming language microSR in HOL. In *Proceedings of the International Higher-Order-Logic Theorem Proving Workshop*, pages 31–44, Vancouver, B.C., August 1993.