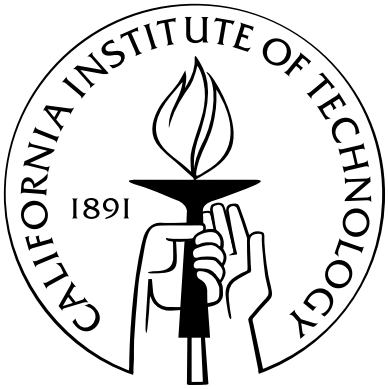
Dynamic UNITY

Thesis by

Daniel M. Zimmerman

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



California Institute of Technology
Pasadena, California

2002
(Defended 27 July 2001)

Acknowledgements

The writing of a doctoral dissertation is a massive undertaking, and simply can't be completed without the contributions and support, academic and otherwise, of many people aside from the author. I take this opportunity to acknowledge the major contributors and supporters.

I am extremely grateful to my primary adviser, Professor K. Mani Chandy, for supporting me during my time at Caltech and for allowing me the freedom to explore research fields that were interesting to me and to also take on some unrelated, and sometimes time-intensive, side projects; little did I know at the beginning that my work would end up using his own as a foundation and springboard. I am also deeply grateful to Professor Jason Hickey for the incredible support, encouragement and assistance he has given me during the late stages of my work. He is my secondary adviser, but denoting him as such understates his contributions. Thanks also to Professor Jehoshua (Shuki) Bruck and Professor Alain Martin, the other two members of my committee, for their participation and feedback on my work.

It has been a privilege to work with the former and current members of Professor Chandy's research group: Dr. Michel Charpentier, Mr. (soon to be Dr.) Roman Ginis, Mr. (soon to be Dr.) Joe Kiniry, Dr. Berna Massingill, Mr. Adam Rifkin, Dr. Eve Schooler, Dr. Paul Sivilotti, and Dr. John Thornley. They have provided encouragement, assistance, feedback, and other support throughout my graduate career, and I thank them all. A special thanks must go to Joe, the one academic colleague with whom I have worked most closely, both inside and outside of the department, during my graduate career. The number of projects he has in the pipeline at any given time never ceases to amaze me, and I'm grateful to have been given the opportunity to participate in some of them. I also thank Dr. Peter Hofstee and Dr. K. Rustan M. Leino who, though they left the research group before I formally joined it in 1996, were formative influences during my undergraduate career in the department.

The administrative assistants play an important role in keeping the department running, and I appreciate the work that they do; in particular, I would like to thank Diane Goodfellow, Jeri Chittum, and Betta Dawson for their assistance during my time as a graduate student.

I've been lucky to have the support of friends, inside and outside Caltech, throughout my graduate career. They've preserved my sanity, provided both distraction and intellectual stim-

ulation, and (usually through judicious application of blunt verbal trauma) given me extra motivation to get "a rnd 2" finishing my graduate work. Guillaume Lessard, who was one of the very first graduate students I met from outside my department, and Gustavo Joseph, with whom I have shared a suite for the last 3 years, are my two closest friends within the Caltech community and have been indispensable. Patrick Hung, Prashant Purohit, and Michel Tanguay, also fellow students, are valued companions as well; I'm sure they would agree that this is an appropriate time to say, "success, gentlemen." Ricky Carson, David Derkits, Robert Duff, and Vale Murthy comprise the regular weekend dinner and gaming crew, with whom I can always be assured (once we manage to decide what to do) of relaxing and having fun. Brian Muzas has also been a dear friend, and I always look forward to seeing him for a stimulating theological debate or building session when I'm back in New Jersey.

For the last 5 years I have lived in Avery House, Caltech's only residence that combines undergraduate, graduate and faculty housing. It's been an interesting experience (to say the least). I'd like to thank all the faculty who have lived in Avery during my time there for their contributions to the community, their advice, and their dinner conversation; especially the Zmuidzinas family, who continue to be good friends.

My parents, Richard and Susan, and my brothers, Matthew and Darin, have been amazingly supportive, both of my graduate studies and of my non-academic projects. Without their encouragement and love, getting to this point would have been impossible. The rest of my family has also been encouraging and supportive throughout my academic career, and I thank them for that.

The research described in this thesis has been supported in part by grants from the Air Force Office of Scientific Research, the CISE Directorate of the National Science Foundation, the Center for Research in Parallel Computing, Parasoft, and Novell Corporation, and by an NSF Graduate Research Fellowship. I thank all these organizations for their support.

This document was prepared in LaTeX2ε, using a few publicly available macro packages and a few of my own. It was typeset directly to Adobe Portable Document Format on a PowerBook running Mac OS X, using the wonderful pdfTeX package as included in version 4.0 of CMacTeX. I thank the author of CMacTeX, Thomas Kiffe, for his swift personal attention to my inquiries about the functionality and use of the software. The typeface used for the main text is Lucida Bright, the sans serif font is Lucida Sans, and the typewriter font is Lucida Sans Typewriter—all three designed by Bigelow & Holmes.

This thesis is available from the Caltech Computer Science Department as technical report number caltechCSTR/2001.006.

Abstract

Dynamic distributed systems, where a changing set of communicating processes must interoperate to accomplish particular computational tasks, are becoming extremely important. Designing and implementing these systems, and verifying the correctness of the designs and implementations, are difficult tasks. The goal of this thesis is to make these tasks easier.

This thesis presents a specification language for dynamic distributed systems, based on Chandy and Misra's UNITY language. It extends the UNITY language to enable process creation, process deletion, and dynamic communication patterns.

The thesis defines an execution model for systems specified in this language, which leads to a proof logic similar to that of UNITY. While extending UNITY logic to correctly handle systems with dynamic behavior, this logic retains the familiar UNITY operators and most of the proof rules associated with them.

The thesis presents specifications for three example dynamic distributed systems to demonstrate the use of the specification language, and full correctness proofs for two of these systems and a partial correctness proof for the third to demonstrate the use of the proof logic.

The thesis details a method for determining whether a system in the specification language can be transformed into an implementation in a standard programming language, as well as a method for performing this transformation on those specifications that can. This guarantees a correct implementation for any specification that can be so transformed.

Contents

List of Definitions

List of Examples

List of Specifications

List of Java Classes

Chapter 1

Introduction

1.1 Motivation

A *distributed system* is a system that consists of multiple communicating processes. A *dynamic* distributed system has the additional characteristic that the processes that make up the system can enter and leave the system while the system is running. Dynamic distributed systems have become increasingly important in recent years, as more computers have been attached to the Internet on a part-time or full-time basis. Most of the core services on the Internet that are used by millions of people daily, including the Domain Name System (DNS) [50, 51] and the Simple Mail Transport Protocol (SMTP) [59], are implemented as large-scale dynamic distributed systems, though most users never see them as such. Popular Internet-based computing projects such as SETI@Home [65], which analyzes signals in an attempt to detect interstellar life, and distributed.net [18], which performs various computations including brute-force encryption cracking and searching for optimal Golomb rulers, are also examples of dynamic distributed systems. These projects allow individual users to participate in huge distributed computations simply by running screen savers or other client programs on their Internet-connected home computers; when a particular computer is connected to the network it communicates with the servers that coordinate the distributed system, and when it is disconnected from the network it continues its computational tasks in isolation.

Both theory and experience have shown that designing correct distributed systems (that is, distributed systems that can be proven to successfully perform the tasks they are designed for) is substantially more difficult than designing correct non-distributed ones. The addition of dynamic behavior makes designing correct systems even more difficult. While many specification and proof methods for distributed systems have been proposed and used to varying degrees of effectiveness, the same is not true of dynamic distributed systems. There is a notable lack of specification and proof techniques for such systems, despite the fact that they are becoming more common.

program division

declare
x, y, z, k: **integer**

initially
x, y, z, k := 0, M, N, 1

assign
z, k := 2 × z, 2 × k if y ≥ 2 × z ~
N , 1 if y < 2 × z
[] x, y := x + k, y − z if y ≥ z

end

Specification 1.1: A UNITY program that implements integer division

The goal of this work is to facilitate the construction of correct dynamic distributed systems, by providing a specification language and proof logic that extends established specification and proof techniques for distributed systems into the dynamic world. In particular, we modify the UNITY formalism, which was introduced by Chandy and Misra [8] for the specification and proof of parallel and distributed programs, to create a new formalism called *Dynamic UNITY* that can be used to specify and reason about dynamic distributed systems. We present examples that illustrate the utility of Dynamic UNITY, and also show how Dynamic UNITY specifications can be implemented as real distributed systems in mainstream programming languages like Java.

1.2 The UNITY Formalism

The UNITY formalism allows for reasoning about concurrent programs in a straightforward fashion. A UNITY program is a set of variables, a set of initialization statements, and a set of multiple assignment statements. An example of a UNITY program is Specification 1.1, which divides the integer M by the integer N and stores the quotient in x and the remainder in y.[1]

The execution of a UNITY program proceeds in the following way: First, the assignments in the **initially** section (if any) are executed. This sets the state variables of the program to their initial values. Then, assignments from the **assign** section are repeatedly chosen and executed according to a weak fairness constraint. In UNITY, the weak fairness constraint ensures that in an infinite execution of a program, every assignment statement in the program is executed

[1] This program is proven to be correct—that is, it is shown that the state variables x and y eventually hold the quotient and the remainder that result from dividing M by N—by Chandy and Misra [8], as the first complete example of a UNITY program with a corresponding proof.

infinitely often. When an assignment statement is executed, its guard (if any) is evaluated, and if it evaluates to **true**, the state variables change according to the assignment statement. In the case of a conditional assignment statement, as in the program above, all the guards are evaluated simultaneously, and the state variables change according to the assignment (if any) whose guard evaluates to **true**. Each assignment statement is executed atomically, so there is no interference among assignment statements that modify the same state variable.

This simple execution model is what makes UNITY a powerful formalism. UNITY has no sequencing operator, and makes no guarantees about the order in which assignments are executed other than the weak fairness guarantee; the execution order of program statements need not be considered explicitly while constructing proofs of UNITY programs. Additionally, the semantics of the multiple assignment operation are well-understood, which means that the behavior of UNITY programs—that by definition consist entirely of atomic multiple assignment operations—is straightforward to analyze.

However, UNITY's simplicity makes it inconvenient for use in specifying dynamic systems. Specifically, UNITY programs have static sets of state variables and static sets of assignments, which means that they can describe systems with dynamic behavior only with great difficulty, by explicitly providing assignments for every possible instance of the dynamic behavior. In addition, UNITY programs are difficult to compose into multiple-program systems, in part because there are no primitive communication operations aside from the multiple assignment statement. While it is certainly possible to build a UNITY system composed of multiple UNITY programs and take advantage of proof reuse, it is extremely difficult, as illustrated in Charpentier and Chandy [11]. The correctness of the program in Specification 1.1 can easily be proven in isolation, but every time it is composed with another UNITY program additional proof steps must be carried out to ensure that no adverse effects arise from the composition.

1.3 Dynamic UNITY

Dynamic UNITY extends the UNITY formalism by adding the concept of processes, new primitives for the creation and destruction of processes, and new primitives and a reliable messaging layer for interprocess communication. It also changes the program notation to one based on binary predicates instead of on assignment statements, similar to the notation of Hehner [23] and of Lamport's Temporal Logic of Actions [35], and eliminates the sharing of variables among multiple programs. These extensions and changes, which will be discussed in more detail later, make it easier to specify and prove the correctness of dynamic distributed systems.

As an example, Specification 1.2 is a Dynamic UNITY program that implements exactly the same algorithm as Specification 1.1. The correctness of this program is proven in a manner sim-

program DivisionModule(M: **integer**, N: **integer**, proc: **process**, mbox: **string**)

declare
x, y, z, k: **integer**

initially
$x = 0 \wedge y = M \wedge z = N \wedge k = 1$

fair-transition
$y \geq 2 \times z \longrightarrow z' = 2 \times z \wedge k' = 2 \times z$
[] $y < 2 \times z \longrightarrow z' = N \wedge k' = 1$
[] $y \geq z \longrightarrow x' = x + k \wedge y' = y - z$
[] $x \times N + y = M \wedge 0 \leq y < N \longrightarrow$ **send**(proc, mbox, x, proc, mbox, y) $\wedge$ **stop**

end

Specification 1.2: A Dynamic UNITY program that implements integer division

ilar to that used for proving the UNITY program. However, while the UNITY program cannot be easily integrated into a larger system, the Dynamic UNITY program has well-defined semantics for such integration: when the division is complete it will send first the quotient and then the remainder as messages to the inbox of process *proc* whose name is contained in *mbox*, and will also stop its own execution (removing itself from the system)[2]. Thus, in any situation where a division needs to be carried out and the quotient and remainder need to be sent to a particular place in the system, an instance of *DivisionModule* can be created with the proper parameters. This particular example would never be used in a real system, as division is typically available as a primitive operation; however, the principle applies to far more complicated programs as well.

By extending the UNITY formalism, we are able to take advantage of the experience accumulated by UNITY practitioners. Much of the high-level logical framework of Dynamic UNITY is essentially identical to that of UNITY, though Dynamic UNITY's underlying execution model differs significantly from UNITY's execution model. Furthermore, Dynamic UNITY's proof logic is based on a small set of fundamental concepts: invariants, variant functions, and leads-to and follows properties. This facilitates both formal reasoning and informal analysis of Dynamic UNITY specifications.

1.4 Contributions

The contributions of this thesis are as follows:

[2]The guard for the transition which performs the message send and stops the process is exactly the predicate which holds at all fixed points of the original UNITY program.

1. A specification language for dynamic distributed systems.
2. A proof logic for dynamic distributed systems.
3. A method for transforming systems from our specification language into an implementation language, so they can be run on actual distributed systems.

1.5 Thesis Structure

The remainder of this thesis is structured as follows:

In Chapter 2, we introduce the Dynamic UNITY formalism. We discuss in some detail the differences between UNITY and Dynamic UNITY and describe the syntax of the Dynamic UNITY language. We also give a brief overview of the execution semantics of Dynamic UNITY systems, including an informal definition of the message-passing layer through which Dynamic UNITY processes communicate.

In Chapter 3, we rigorously define the execution model for Dynamic UNITY systems and the Dynamic UNITY message-passing layer, and introduce a logic for the verification of Dynamic UNITY specifications. We adapt the temporal operators commonly used in proving the correctness of UNITY programs for use in proving the correctness of Dynamic UNITY specifications.

In Chapters 4 and 5, we present two example Dynamic UNITY systems—a prime number sieve, which results in the creation of an infinite number of communicating processes, and a dynamic single resource mutual exclusion system, where consumers of the single resource can enter and leave the system at any time during their execution. We also carry out complete proofs of correctness for these systems.

In Chapter 6, we present an example Dynamic UNITY system that implements a solution to a dynamic version of the drinking philosophers problem. We also carry out a partial proof of progress for our solution. The safety properties of this solution are straightforward and similar to those of the single resource mutual exclusion system.

In Chapter 7, we discuss techniques for the implementation of Dynamic UNITY specifications on actual computer systems. As an example of one such implementation technique, we formulate a method for the direct translation of many Dynamic UNITY specifications into Java code.

In Chapter 8, we outline some related work and compare our research to other specification and proof methods for distributed systems.

In Chapter 9, we present a summary of our results and a discussion of the applicability of these results. We also discuss potential future research directions.

Chapter 2

Dynamic UNITY

In this chapter, we introduce the Dynamic UNITY formalism, which allows us to reason about algorithms and protocols in which the sets of participating processes change over time. By extending the familiar UNITY formalism, we are able to take advantage of many UNITY proof rules and techniques while reasoning about dynamic systems that cannot be adequately described using UNITY.

We first detail the changes and extensions to the original UNITY formalism that allow Dynamic UNITY to encompass dynamic systems; then, we introduce the Dynamic UNITY notation and briefly discuss the execution semantics of Dynamic UNITY systems.

2.1 Extending UNITY to Dynamic Systems

We have previously given an overview of the UNITY formalism, including a brief description of its execution model. Changes to the execution model, as well as to the proof logic and the specification language itself, are necessary to adapt UNITY to handle systems where processes can be created and destroyed at runtime. In order to make this task more manageable, we restrict our attention to systems that satisfy the following constraints:

1. Each process has access only to its own state—that is, there is no direct sharing of variables among the processes.

2. Processes communicate only via asynchronous message passing.

3. Any process can create new processes, and any process can destroy itself, but no process can destroy other processes.

These constraints are not chosen arbitrarily—each helps to make the tasks of designing and proving the correctness of a dynamic distributed system easier. The first eliminates any possibility that processes can directly interfere with each others' operation, allowing for both

modular reasoning (proof reuse) and modular system construction; the second restricts interprocess communication to a single well-understood mechanism, which simplifies system design; and the third simplifies proof obligations by eliminating the possibility that a running process will be destroyed at an unexpected or inappropriate time.

Each constraint is enforced by specific changes to the UNITY formalism. We eliminate the notion of shared variables entirely, since we are only considering systems with no shared state. We add reliable first-in first-out message passing as a primitive of the language and create new operations to manipulate messages in various ways. We also add the notion of programs and processes by changing the formalism in a fundamental way—instead of a single program, a Dynamic UNITY system consists of multiple programs that are instantiated as processes and that can halt their own execution. We can then prove properties about individual programs independent of the behavior of other programs in the system.

In addition to these changes, we also change the notation for Dynamic UNITY specifications: instead of guarded parallel assignment statements, we use guarded binary predicates to represent state transitions. This change provides more flexibility for program designers, enabling them to focus directly on the result of each transition rather than on the precise set of assignment statements needed to make that result happen. However, it also makes it much easier to create Dynamic UNITY specifications that are not implementable on actual computer systems, a topic we will discuss further in Chapter 7.

2.2 Dynamic UNITY Notation

We describe the notation of Dynamic UNITY using BNF. Nonterminal symbols are italicized, and terminal symbols are in plain or boldface type. "(X)*" denotes a syntactic unit X that may be instantiated zero or more times, "$(X)^+$" denotes a syntactic unit X that may be instantiated one or more times, "$(X)^1$" denotes a syntactic unit X that may be instantiated one time or not instantiated at all.

During the notation description, we enumerate certain conditions under which particular constructs are considered malformed. A Dynamic UNITY system with malformed constructs can be syntactically legal, but we make no guarantees about the behavior of systems with malformed constructs. A system that contains no malformed constructs is considered well-formed; a proof of well-formedness is one of the obligations for any correctness proof of a Dynamic UNITY system.

2.2.1 Program Structure

program-section ⟼ (**program** | **initial-program**) *program-name* ((*parameter-list*))[1]
(**type** *type-section*)[1]
(**declare** *declare-section*)[1]
(**always** *always-section*)[1]
(**initially** *initially-section*)[1]
(**fair-transition** *transition-section*)[1]
(**unfair-transition** *transition-section*)[1]
end

parameter-list ⟼ *parameter-name*: *external-type-name*
(, *parameter-name*: *external-type-name*)*

A program can start with either the keyword **program** or the keyword **initial-program**. The latter indicates that the program will be the first one instantiated when system execution begins. A system's initial program is usually a setup program that instantiates one or more other programs to bootstrap the system. There is always exactly one initial program in a well-formed system.

A *program-name* is any string of letters and digits, with the restriction that no two programs in the same system may have identical names. A *parameter-name* is any string of letters and digits, with the restrictions that no two parameters declared in a program may have identical names and that no parameter declared in a program may have the same name as a variable or definition declared in that program. An *external-type-name* is a primitive data type (such as **integer** or **set**), the name of a data type declared in the *type-section* (described in Section 2.2.2) of the system, or the name of another program declared in the system.

The program execution starts in a state where the set of parameters is as specified at process instantiation (described in Section 2.2.8.1), the set of variables is as declared in the *declare-section* (described in Section 2.2.3), the set of definitions is as declared in the *always-section* (described in Section 2.2.4), and the initial states of all variables are as specified in the *initially-section* (described in Section 2.2.5).

The current definition of Dynamic UNITY does not allow for hierarchical program structure. We discuss the possibility of adding this capability to Dynamic UNITY in Chapter 9.

2.2.2 Type Section

type-section ⟼ (*declared-type-name*: *type-specifier*)+

type-specifier ⟼ *array-type* | *primitive-type* | *record-type* | *sequence-type* | *set-type*

array-type ⟼ **array**^{dim} {*type-name* | *type-specifier*}

primitive-type ⟼ **any** | **boolean** | **inbox** | **integer** | **process** | **real** | **string**
record-type ⟼ **record** {*field-name*: (*type-name* | *type-specifier*)
(, *field-name*: (*type-name* | *type-specifier*))*}
sequence-type ⟼ **sequence** {*type-name* | *type-specifier*}
set-type ⟼ (**multiset** | **set**) {*type-name* | *type-specifier*}

A *declared-type-name* is any string of letters and digits, with the restrictions that no two types declared in the same program may have identical names and that no type declared in a program may have the same name as a primitive type, a type declared in the system containing that program, or a variable declared in that program. A *type-name* is a primitive data type, the name of a data type declared in the program's type-section or the system's type-section, or the name of another program declared in the system. A *field-name* is any string of letters and digits, with the restrictions that no two fields declared in the same type may have the same name and that no field may be named "type" (**type** is a Dynamic UNITY operation that allows a process to determine the type of an object it receives through message passing; it is described in Section 2.2.8.4). The *dim* superscript used with **array** is a positive integer that determines the array's dimensionality. The any type allows for the construction of sets, arrays, and sequences that can contain elements of any primitive or declared type.

Dynamic UNITY's primitive types support the expected range of operations: individual elements of arrays and sequences can be accessed using "[]" syntax, fields of records can be accessed using "." syntax, and sets and multisets can be used with the operators normally associated with mathematical sets. In addition, a binary concatenation operator ($\bowtie$) generates sequences from other sequences or elements.

Some of Dynamic UNITY's primitive types have "special" values: the empty set is denoted by $\varnothing$, the empty sequence is denoted by Λ, and the null process, a value given to a process reference to indicate that it does not point to a process, is denoted by $\perp$.

Dynamic UNITY's declared types are not types in the sense of type theory. Instead, they are tags that exist mainly for structuring and to facilitate message-based communication. They are most commonly used to distinguish between multiple message types that may be received by a given process when the reaction of the process is dependent on the type of a received message.

Example 2.1 (Type declarations)

1. A user record containing a username, a password, and a unique identification number:

   UserRecord: **record** {name: **string**, password: **string**, UID: **integer**}

2. A set of user records:

   UserRecordSet: **set** {UserRecord}

2.2.3 Declare Section

declare-section $\longmapsto$ (*variable-name* (, *variable-name*)*: *type-name*)$^+$

A *variable-name* is any string of letters and digits, with the restrictions that no two variables declared in a program may have identical names, that no variable declared in a program may have the same name as a parameter or definition declared in that program, and that no variable may have the same name as a type declared in the type-sections of its program or the system. A type-name is as described in Section 2.2.2.

The declare-section contains the variables whose values can be changed by the program during execution. Initial values for these variables are specified in the initially-section (described in Section 2.2.5). Any variable for which an initial value is not specified in the initially-section is implicitly initialized to a canonical default value for its specified type, as follows: anys are initialized to the empty set, arrays (of unspecified length), sequences and sets are initially empty, processes are initialized to the empty process, booleans are initialized to **false**, integers and reals are initialized to 0, and strings are initialized to the empty string. Inboxes are implicitly assigned names equivalent to their variable names (so an inbox declared as "myInbox: **inbox**" would be named "myInbox"). Implicit initialization is recursive; for instance, all the elements of an array of integers with specified length are initialized to 0. Note that it is not possible to change the name of an inbox once it has been initialized, so any inboxes which need to have names other than their variable names *must* have these names specified in the initially-section.

Variable names may appear primed (myVariable′) in the transition-section (described in Section 2.2.6), but may not appear primed anywhere else in a program. As in the notation of Hehner [23] and in Lamport's Temporal Logic of Actions [35], primed variable names represent the variables after a state transition, while unprimed variable names represent the variables before a state transition. Primed and unprimed variable names can be used together to specify binary predicates on the program variables.

Example 2.2 (Variable declarations)

1. Two sets of user records (explicit):

   privilegedUsers, unprivilegedUsers: **set** {UserRecord}

2. Two sets of user records (using the type defined in Example 2.1):

   privilegedUsers, unprivilegedUsers: UserRecordSet

3. A sequence where the elements can be of any type:

   unrestrictedSequence: **sequence** {**any**}

4. An array of sequences of integers:

   sequencesOfIntegers: **array** {**sequence** {**integer**}}

2.2.4 Always Section

always-section $\longmapsto$ *definition-name* $\triangleq$ *expression*
(; *definition-name* $\triangleq$ *expression*)*

A *definition-name* is any string of letters and digits, with the restrictions that no two definitions declared in a program may have identical names and that no definition declared in a program may have the same name as a parameter or a variable declared in that program. An *expression* is an arithmetic or logical expression in one or more of the variables, parameters and definitions declared in the program, and may also contain certain Dynamic UNITY operations (described in Section 2.2.8). It may not contain primed variables.

The always-section is used to create definitions, which are functions of program variables and parameters. Definition types are implicitly specified by the expressions associated with them. A definition may be used in subsequent definitions, initialization expressions, and transition expressions. Every appearance of a definition can be considered a "macro" for the expression associated with it. Definitions are considered to be defined in the order in which they appear (the use of a semicolon as the separator for definitions reinforces this notion of sequencing). An always-section that contains cyclical definitions is considered malformed.

The value of a definition may not be changed as part of an initialization expression (see Section 2.2.5) or transition expression (see Section 2.2.6).

Example 2.3 (Well-formed always-sections)

1. A Boolean flag that is **true** if and only if the privileged user set is empty:

   privilegedUsersIsEmpty $\triangleq$ |privilegedUsers| = 0

2. Two Boolean flags, one of which is **true** if and only if the privileged user set is empty and the other of which is **true** if and only if both user sets are empty:

   privilegedUsersIsEmpty $\triangleq$ |privilegedUsers|=0;
   noUsers $\triangleq$ (privilegedUsersIsEmpty $\wedge$ |unprivilegedUsers|=0)

3. An integer that holds the total number of user records:

   numberOfUsers $\triangleq$ |privilegedUsers| + |unprivilegedUsers|

EXAMPLE 2.4 (MALFORMED ALWAYS-SECTIONS)

1. The simplest possible cycle:

   flag $\triangleq$ $\neg$flag

2. A more complex cycle:

   definitionA $\triangleq$ parameterOne $\leq$ parameterTwo;
   definitionB $\triangleq$ ((parameterThree $\geq$ parameterFour) $\wedge$ $\neg$definitionD);
   definitionC $\triangleq$ definitionA $\vee$ definitionB;
   definitionD $\triangleq$ $\neg$definitionC $\vee$ (parameterFour $\geq$ parameterOne)

2.2.5 Initially Section

initially-section $\longmapsto$ *initialization-guard* $\longrightarrow$ *initialization*
($\|$ *initialization-guard* $\longrightarrow$ *initialization*)*

An *initialization-guard* is a predicate on the program parameters and definitions that depend only on the program parameters. An *initialization* is a (unary) predicate on the program variables, parameters and definitions that may contain variable names and Dynamic UNITY operations (described in Section 2.2.8). This predicate constrains the initial state of the program. An initialization written with no guard is considered to have **true** as its guard.

The initially-section is used to specify the initial values of variables declared in the declare-section (described in Section 2.2.3). All guards are evaluated simultaneously at the beginning of initialization; for each guard that holds when evaluated, the corresponding initialization predicate is guaranteed to hold after initialization. If multiple guards are not mutually exclusive, their initialization predicates must not specify different values for the same variable. In addition, the inclusion of a process instantiation (see Section 2.2.8.1) in one or more initializations incurs an obligation to prove that no infinite recursion of initially-sections is possible. An initially-section that contains conflicting initializations or causes an infinite recursion is considered malformed.

EXAMPLE 2.5 (WELL-FORMED INITIALIZATIONS)

1. Nondeterministic initialization of an integer variable to one of two values:

   anInteger = 42 $\vee$ anInteger = 731

2. Nondeterministic initialization of an integer to any positive value:

   anInteger > 0

3. Initialization of privileged and unprivileged user sets from passed parameters only if a specific flag is passed:

$$\begin{aligned} \text{initializationFlag} &\longrightarrow \text{privilegedUsers} = \text{initialPrivilegedUsers} \wedge \\ &\qquad\quad \text{unprivilegedUsers} = \text{initialUnprivilegedUsers} \\ \| & \\ \neg\text{initializationFlag} &\longrightarrow \text{privilegedUsers} = \varnothing \wedge \text{unprivilegedUsers} = \varnothing \end{aligned}$$

4. Initialization of an inbox, assigning it the name "aNewName":

   myInbox = **inbox**("aNewName")

Example 2.6 (Malformed initializations)

1. Malformed nondeterministic initialization of an integer variable to one of two values:

$$\begin{aligned} &\textbf{true} \longrightarrow \text{anInteger} = 42 \\ \| & \\ &\textbf{true} \longrightarrow \text{anInteger} = 731 \end{aligned}$$

2. Malformed initialization of the privileged and unprivileged user sets from passed parameters only if a specific flag parameter holds:

$$\begin{aligned} \text{initializationFlag} &\longrightarrow \text{privilegedUsers} = \text{initialPrivilegedUsers} \wedge \\ &\qquad\quad \text{unprivilegedUsers} = \text{initialUnprivilegedUsers} \\ \| & \\ \textbf{true} &\longrightarrow \text{privilegedUsers} = \varnothing \wedge \text{unprivilegedUsers} = \varnothing \end{aligned}$$

2.2.6 Transition Section

$$\begin{aligned} \textit{transition-section} &\longmapsto \textit{transition-list} \\ \textit{transition-list} &\longmapsto \textit{transition-list-element} \mid \textit{quantified-transition-list} \\ &\quad ([]\ \textit{transition-list-element} \mid \textit{quantified-transition-list})^* \\ \textit{quantified-transition-list} &\longmapsto \langle [] \ \textit{variable-list} \mid \textit{ranges} \triangleright \textit{transition-list} \rangle \\ \textit{transition-list-element} &\longmapsto (\textit{variable-list}:)^1\ \textit{transition-guard} \longrightarrow \textit{transition-predicate} \\ \textit{variable-list} &\longmapsto \textit{variable-name}\ (, \textit{variable-name})^* \end{aligned}$$

A *transition-guard* is a predicate on the program parameters, variables and definitions describing only a single program state (that is, containing no primed variables). A *transition-predicate* is a predicate on the program parameters, variables and definitions that may contain

primed variable names and some special operations detailed in Section 2.2.8. A transition-predicate is therefore a binary predicate on the pre- and post- states of the transition. A transition written with no guard is considered to have **true** as its guard. Note that the ⟶ symbol in a transition is not a logical implication, but rather a separator between the guard and the transition (as found in the guarded commands of CSP [26]).

Each transition has an optional *variable-list* (the *transition variables list*) associated with it. This list specifies all the variables whose values can be changed as a result of the transition's execution. Variable names in the list that do not appear in the program's declare-section denote temporary variables that exist only in the scope of the transition. Every variable whose primed name appears in a transition, either explicitly or as part of a messaging operation, is considered to be in the transition variables list for that transition regardless of whether it is listed. A transition variables list consisting only of variables whose primed names appear in the transition may be omitted entirely.

A quantified transition contains a variable-list (the *bound variables list*), a *ranges* predicate that constrains the ranges over which the bound variables are quantified, and a transition-list. The bound variables list of a quantified transition may not contain any variable names that appear in the program's declare-section, and variables contained within the bound variables list may not appear primed in, or as part of the transition-variables list of, any transition in the quantified transition's transition-list. The ranges predicate may not contain primed variables.

The transition-sections comprise the "body" of the program. They determine all possible state transitions that may take place during the program's execution. A transition is *satisfiable* if, for every program state where its guard holds, its transition-predicate can be satisfied by establishing a post-state in which every variable whose value differs from that in the pre-state appears in the transition-variables list. A transition's satisfiability is therefore independent of program executions, as it depends only on the guard, the transition-predicate, and the types of all variables in the transition-variables list.

In addition to satisfiability, we require that the quantification(s) of a Dynamic UNITY transition be countable. An unsatisfiable Dynamic UNITY transition, or one with an uncountable quantification, is considered malformed. While we do not explicitly disallow malformed transitions, it is important to note that we make no guarantees about the behavior of Dynamic UNITY systems with malformed transitions.

Clearly, it is not always practical (or even possible) to tell whether or not a particular transition is unsatisfiable, because such a determination may itself involve an intractable or undecidable computation. However, we make it a proof obligation that all transitions must be explicitly proven satisfiable. Thus, it is possible to write a Dynamic UNITY program that contains only satisfiable transitions, but not be able to prove the correctness of that program because it isn't

possible to prove the satisfiability of the transitions.

In addition, there is an important distinction between malformed transitions and *uncomputable* transitions. An uncomputable transition is one where the guard or postcondition is uncomputable; for example, the guard, "X is a nonterminating Turing machine," is uncomputable, because computing it would require a solution to the halting problem. However, we can still make guarantees about uncomputable transitions provided that they are satisfiable (such as, "X is a nonterminating Turing machine $\longrightarrow Y' = X$"). We can therefore prove the correctness of Dynamic UNITY systems which contain uncomputable transitions, even though we can never implement these systems on real computers. The differences between the Dynamic UNITY systems we can prove the correctness of and the ones we can actually implement are discussed further in Chapter 7.

Example 2.7 (Well-formed transitions)

1. Increment an integer variable by 1 if its value is less than the value of integer parameter MAXINT:

   $\text{anInteger} < \text{MAXINT} \longrightarrow \text{anInteger}' = \text{anInteger} + 1$

2. Increment every integer variable in the set bigSet:

   $\text{bigSet}' = \{i \mid i \in \text{bigSet} \rhd i + 1\}$

Example 2.8 (Malformed transitions)

1. Increment an integer variable by 1 if its value is less than the value of integer parameter MAXINT, with an additional constraint:

   $\text{anInteger} < \text{MAXINT} \longrightarrow \text{anInteger}' = \text{anInteger} + 1 \wedge \text{anInteger}' < \text{MAXINT}$

2. Perform any computation, by establishing the postcondition **false**:

   false

Example 2.9 (Quantified transitions)

1. Fill in the values of an identity matrix of dimension N:

$$\begin{aligned}\langle [] \; i, j \mid 0 \le i < N \wedge 0 \le j < N \rhd \\ \quad i = j \quad &\longrightarrow \text{matrix}'[i,j] = 1 \\ [] \quad i \neq j \quad &\longrightarrow \text{matrix}'[i,j] = 0 \; \rangle\end{aligned}$$

2. Remove all the integers from set S, one at a time (the **type** operation is discussed in Section 2.2.8.4):

   $\langle [] \; i \mid i \in S \wedge i.\textbf{type} = \textbf{integer} \rhd S' = S \setminus \{i\} \rangle$

2.2.7 System Structure

system ⟼ **system** *system-name*
(**type** *type-section*)[1]
(*program-section*)[+]
end

A *system-name* is any string of letters and digits. There are no restrictions on this string; it is currently used only for referring to the system in proofs and discussions. The type-section, program-section and initially-section are exactly as described previously.

A Dynamic UNITY system contains one or more component programs, with at most one designated as an initial program (as described in Section 2.2.1). A system containing no initial program is legal, but such a system will have no transitions and will therefore never do anything.

2.2.8 Operations

In addition to variable state changes, Dynamic UNITY supports operations for process instantiation, process destruction, messaging, and introspection on variable types. In this section, we describe the syntax of these operations and the contexts in which each can be used.

2.2.8.1 Process Instantiation

Dynamic UNITY processes are instantiated with the **new** operation, which has the following syntax:

new-operation ⟼ *reference-name* = **new** *program-name* **(***passed-parameters***)**
passed-parameters ⟼ *parameter* (, *parameter*)*

A *reference-name* is the name of a variable of type *program-name*, which will contain a reference to the instantiated process as a postcondition of the new operation. A program-name is exactly as described in Section 2.2.1, and a *parameter* is any variable or value. The types of the parameters in the *passed-parameters* list must match the types of the parameters in the parameter-list of the program being instantiated.

All parameters in Dynamic UNITY are passed by value. The values held by the parameters at the execution of the new operation are copied locally for the new process, and are treated as constants by the new process during its execution. The new process' initially-section is executed immediately as part of the new operation, and its transitions are available for execution (that is, they can be chosen by the scheduler) immediately after the new operation.

Process instantiations can occur in both the initially-section and the transition-sections of a program. Therefore, the reference-name can be either primed (a variable assignment in a transition-section) or unprimed (an initialization).

2.2.8.2 Process Destruction

A Dynamic UNITY process can halt its execution with the **stop** operation. The syntax for this operation is the keyword **stop**, used as a conjunct or disjunct in a transition.

When a **stop** operation is executed, the process's transitions are removed from the system and no more changes to the process's state (excluding the states of its mailboxes) ever occur. A process that executes a **stop** is effectively destroyed. All messages sent by the process, including those sent in the same transition as the **stop**, are delivered by the message-passing system just as they would have been without the process destruction. Similarly, all messages sent to the process by other processes in the system are delivered to its inboxes, even though the process will never be able to read them.

When **stop** is used as a disjunct in a transition, the process may or may not be destroyed after that transition is completed; such nondeterministic process destruction can be used to simulate process failures.

In the Dynamic UNITY formalism, the state of a destroyed process remains in the system perpetually. This facilitates the construction of proofs that depend on the final states of destroyed processes. However, in an actual implementation, a destroyed process's state would most likely be removed from memory for space and efficiency considerations.

2.2.8.3 Messaging

Dynamic UNITY contains a messaging system that uses incoming and outgoing message queues. The Dynamic UNITY type **inbox** implements incoming message queues, and support operations that allow processes to receive and inspect incoming messages. Each Dynamic UNITY process has a single outgoing message queue (called an *outbox*) that it can use to send messages to other processes. The following describes the internal structure of inboxes and outboxes, as used in proofs of correctness; however, Dynamic UNITY transitions cannot read or modify any of this internal structure directly, and can only interact with the messaging system using the messaging operations described later in this section. This restriction prevents Dynamic UNITY programs from relying on any specific behavior of the messaging system (such as its interleaving with the execution of regular Dynamic UNITY transitions) other than the fact that it establishes point-to-point first-in-first-out channels.

Inboxes are treated as named sequences of records. They have three associated attributes:

name, a string that uniquely identifies the inbox within its process; *length* (usually abbreviated *len*), the number of messages in the inbox; and *count* (usually abbreviated *cnt*), the index of the next message to be read.

The outbox for each Dynamic UNITY process is also treated as a sequence of records, and has one attribute associated with it—*length* (usually abbreviated *len*), the number of messages that have been sent by the process. An additional attribute, *count* (usually abbreviated *cnt*), is derived from the *delivered* fields of the message records in an outbox (described below); it is the number of *delivered* fields that have the value **true**, which is exactly the number of messages in the outbox that have been delivered.

Inboxes and outboxes are sequences of unbounded length, preserving the entire message history for all processes in the system. This facilitates the construction of proofs that depend on message histories and message ordering. However, in an actual implementation, message histories would most likely be kept small for space and efficiency considerations.

Each record contained in an inbox or outbox has multiple fields: records in inboxes have 2 fields, while records in outboxes have 4. These fields are defined as follows:

- The *process* field (usually abbreviated *proc*) appears in both inbox and outbox records. In an inbox record, it contains a reference to the process that sent the message contained in that record. In an outbox record, it contains a reference to the process that will receive (or has received) the message contained in that record.

- The *mailbox* field (usually abbreviated *mbox*) appears only in outbox records. It contains the name of the inbox to which the message contained in the record will be (or has been) delivered.

- The *message* field (usually abbreviated *msg*) appears in both inbox and outbox records. In an inbox record, it contains the message data that was received. In an outbox record, it contains the message data that will be (or has been) sent.

- The *delivered* field (usually abbreviated *del*) appears only in outbox records. It contains a Boolean value that indicates whether or not the message contained in the record has been delivered to its destination (it is **false** if the message has not been delivered, and **true** if it has).

The attributes of outboxes and inboxes are all updated atomically to reflect the new state of the messaging system when a message delivery occurs. Since these attributes are not observable by Dynamic UNITY programs, their updates do not actually have to be atomic in an implementation of a Dynamic UNITY system as long as the messaging system correctly im-

plements reliable point-to-point first-in first-out channels. They are used only in proofs of correctness, and in our detailed specification of the Dynamic UNITY execution model.

Dynamic UNITY programs use messaging operations to access the mailbox data structures we have described, and cannot access these structures by any other means. Outboxes support a single operation that allows for the sending of messages (or sequences of messages). Inboxes support four operations that allow for the detection of available messages, reading of the next available message, removal of the next available message, and reading of an inbox's name. The syntax and an overview of the semantics for these operations are described in the remainder of this section. Semantics for these operations are described in detail with the semantics of the messaging system in Section 2.3.2.

Send The **send** operation on an outbox causes a message (sequence of messages) to be sent to a destination inbox (sequence of destination inboxes). Its syntax is as follows:

send-operation	$\longmapsto$	**send** (*message-send-list*)
message-send-list	$\longmapsto$	*message-send* (, *message-send*)*
message-send	$\longmapsto$	*process-reference*, *inbox-name*, *message*

A *process-reference* is a variable or parameter that contains a reference to a process. An *inbox-name* is a string, which is the name of the inbox to which the associated *message* will be sent. A message is any value or variable.

Semantically, a send operation is a manipulation of the outbox sequence that causes one or more appropriate outbox records to be appended to the sequence. When a send operation includes more than one *message-send*, the message records are appended to the outbox sequence in the order in which they are listed. Quantifier-like syntax may be used to send messages to a set of inboxes; this is shown in the example below. The quantification of a message send must be finite—a message send with an infinite quantification is considered malformed. While we do not explicitly disallow malformed message sends, we make no guarantees about Dynamic UNITY systems that contain malformed message sends.

The send operation can be used in both the initially-section and the transition-sections of a program. If the send operation is used in two or more different initializers of a program, no ordering guarantee is provided with respect to the sends (they are equally likely to occur in any of the possible orderings).

Example 2.10 (Quantified Message Sends)

1. Send a different message of type "MessageType" to the inbox named "in" belonging to each process in set P:

   send $(\langle , p \mid p \in P \rhd p, \text{“in”}, \text{MessageType}(p)\rangle)$

2. Send identical messages to each inbox whose name is contained in set I belonging to each process in set P:

   send $(\langle , p, i \mid p \in P \wedge i \in I \triangleright p, i, \text{theMessage} \rangle)$

Probe The **probe** operation on an inbox evaluates to the Boolean value **true** if there is a message that has been placed in the inbox but has not been advanced past with the **advance** operation, and evaluates to the Boolean value **false** otherwise. Its syntax is *inbox-var*.**probe**, where *inbox-var* is the inbox variable to probe. The probe operation is shorthand for a comparison between the inbox's current message counter and the number of messages in the inbox; it can therefore be used anywhere a predicate that makes such a comparison can be used.

Current The **current** operation on an inbox allows read-only access to the current message in the inbox (as determined by the inbox's *cnt* attribute). Its syntax is *inbox-var*.**current**.

The current operation can be used on the right-hand side of assignments, as well as in any non-assignment predicate. It is handled exactly as a record containing the inbox record fields described above: the actual message data is *inbox-var*.**current**.msg, and the reference to the sending process is *inbox-var*.**current**.proc.

If no messages have ever been placed in the inbox, or if the inbox has been advanced past its last message, the current operation returns a record containing the null set as its message data and the null process as its sender.

Advance The **advance** operation on an inbox advances the inbox to the next message. Its syntax is *inbox-var*.**advance**. The advance operation is shorthand for an increment of the inbox's current message counter; it can therefore be used anywhere a predicate that increments a variable can be used.

Name The **name** operation on an inbox allows read-only access to the inbox's name as a string. Its syntax is *inbox-var*.**name**. It can be used anywhere a string (or a variable of type string) can be used.

2.2.8.4 Introspection

In addition to the process and messaging operators, Dynamic UNITY supports operations that perform limited introspection on entities (such as messages received in an inbox). These operations allow a process to determine the type of an entity, as well as its length (for arrays or sequences) or cardinality (for sets). There is also an operation that allows a process to obtain a reference to itself.

Type The **type** operation allows a process to determine and act on the type of a Dynamic UNITY entity. Its syntax is *entity*.**type**, where *entity* is a variable or an entity assignable to a variable (such as *myInbox*.**current**.msg). It is typically used in comparisons with type names or type specifiers for conditional message receives, as shown in the examples below, but can be used elsewhere. Type names used in a comparison must be the names of either primitive types or types declared in the scope of the comparison; the elimination of potential confusion about the interpretation of type comparisons is the main reason why types and variables are not allowed to have identical names.

EXAMPLE 2.11 (TYPICAL USAGE OF THE TYPE OPERATION)

1. Receive a message into variable *theMessage* only if the message data is of type "myDesiredType."

   myInbox.**probe** $\wedge$ myInbox.**current**.msg.**type** = myDesiredType $\longrightarrow$
   (theMessage$'$ = myInbox.**current**.msg) $\wedge$ myInbox.**advance**

2. Receive a message into variable *theMessage* only if the message data is of the same type as the variable.

   myInbox.**probe** $\wedge$ myInbox.**current**.msg.**type** = theMessage.**type** $\longrightarrow$
   (theMessage$'$ = myInbox.**current**.msg) $\wedge$ myInbox.**advance**

Length The **length** operation allows a process to determine the current number of elements in an array or a sequence, or the length of a string; it cannot be used on mailboxes. Its syntax is *entity*.**length**, where *entity* is a variable holding a value of an array, sequence or string type. The analogous operation for a set, cardinality, is performed using the standard mathematical syntax for sets ($|set|$). The length (or cardinality) operation results in an integer constant, and it can be used anywhere an integer constant can be used.

This The **this** operation allows a process to obtain a reference (of type **process**) to itself. Its syntax is the keyword **this**, used in a transition within the process, and it can be used anywhere a value of type **process** can be used.

2.3 Dynamic UNITY Semantics

We now describe the semantics of Dynamic UNITY, paying particular attention to the areas where they differ from the semantics of the original UNITY formalism.

2.3.1 Execution Model

Dynamic UNITY's execution model is similar to UNITY's execution model, in that both atomically execute a single statement at a time from a set of statements in a weakly fair manner. However, this is where the similarities end. While a UNITY program has a static set of guarded assignment statements that are all subject to weak fairness, a Dynamic UNITY system has a dynamic set of processes, where each process has a set of guarded transitions, some of which are subject to weak fairness and some of which are not. By including the ability to create transitions that are not subject to fairness constraints, Dynamic UNITY can more accurately model real distributed systems in which particular events (such as requests in a resource allocation system) may not occur in a fair manner. The definition of weak fairness itself is different in Dynamic UNITY, because the set of transitions available in the system can change. We first define a weakly fair transition, and then introduce our definition for weak fairness:

Definition 2.1 (Weakly Fair Transition) *A weakly fair transition in a Dynamic UNITY system is an instantiation of a transition-statement within the fair-transition section of a Dynamic UNITY program, specified by the process to which it belongs and any quantifying terms used to generate it.*

This definition means that two different instantiations of the same Dynamic UNITY program have two different sets of weakly fair transitions. It also means that a quantified transition is considered not as a single weakly fair transition, but as a number of weakly fair transitions (depending on the range of the quantification). Given this definition for a weakly fair transition, the definition of weak fairness for Dynamic UNITY systems is as follows:

Definition 2.2 (Weak Fairness) *In every computation of a Dynamic UNITY system, every weakly fair transition is infinitely often either selected or not present in the system.*

This definition implies that a transition that remains in the system forever will execute infinitely often. It does not, however, imply that a transition that is merely present in the system infinitely often will execute infinitely often. Weak fairness will be discussed in more detail when we formalize the execution model in Chapter 3; the following is an informal execution model for Dynamic UNITY systems:

In each system step, at most one state transition occurs. Transitions are selected from the *fair-transition* and *unfair-transition* sections (described in Section 2.2.6) of the processes currently running in the system. When a transition is selected, its precondition is evaluated. If it evaluates to true, the transition is executed atomically and its postcondition holds at the end of this execution, and if it evaluates to **false**, the state of the system is left unchanged. If a transition is selected whose precondition evaluates to **true** and whose postcondition is

system ExampleSystem

initial-program ExampleSystemComponent

declare
theSet: **set** {**integer**}

initially
theSet = $\emptyset$

fair-transition
(1) **true** $\longrightarrow \langle \exists\, i \mid i \notin \text{theSet} \rhd \text{theSet}' = \text{theSet} \cup \{i\} \rangle$
(2) [] **true** $\longrightarrow$ **stop**
(3) [] $\langle$[]i | i $\in$ theSet $\rhd$ i $\geq$ 731 $\longrightarrow$ p: p′ = **new** ExampleSystemComponent$\rangle$

end

end

Specification 2.1: An example system used to illustrate a changing transition set

unsatisfiable at that point in the execution, we cannot say anything about the state of the system which results. It is therefore important, when constructing Dynamic UNITY systems, to ensure that they contain no malformed transitions.

In an infinite execution of the system, every transition in the fair-transition section of each running program is guaranteed to be selected in a manner consistent with the weak fairness definition. No guarantee is made about how often each transition in the unfair-transition section of each program is selected. A system step can result in the addition or removal of processes from the system, as well as the addition or removal of quantified transition instances due to state variable changes. We now present a simple system, and trace an example execution to illustrate how the set of transitions in the system changes.

Example 2.12 (An illustration of a changing transition set)
The simple Dynamic UNITY system of Specification 2.1 contains processes that have a single set as a state variable, and that can create new processes and destroy themselves. It illustrates the changing nature of the transition set in a Dynamic UNITY system.

This system consists of one type of process, which has only one state variable: a set of integers. Its three transitions have the following effects:

1. Adds an integer to the set, ensuring that the added integer was not already in the set. We note that this addition is not fair, even though the transition is a fair transition; while we guarantee that this transition executes infinitely often, we do not guarantee that every integer i will eventually become part of *theSet*.

2. Stops the process.

3. Expands to a transition for every integer in the set, which instantiates a new process if the integer is greater than 731. For example, if the set consists of the integers 1013, -5, and 0, this would expand to the following three transitions (recall that the "p:" syntax defines a temporary variable in the scope of its associated transition):

$$\begin{aligned} 1013 \geq 731 &\longrightarrow \text{p: } p' = \textbf{new } \text{ExampleSystemComponent} \\ -5 \geq 731 &\longrightarrow \text{p: } p' = \textbf{new } \text{ExampleSystemComponent} \\ 0 \geq 731 &\longrightarrow \text{p: } p' = \textbf{new } \text{ExampleSystemComponent} \end{aligned}$$

We denote the transitions of the system by a process identifier (A, B, C, ...), a number (taken from the program above), and, when necessary, a subscript indicating a particular instantiation of a quantified transition. We subscript state variables with their corresponding process identifiers for clarity, since we are presenting the system's transitions as one combined set. Using these conventions, one legal execution of this system can be described as follows:

1. Initialization: A process (to which we assign the identifier A) is instantiated from the system's initial program. A's state variable *theSet* is initialized to the empty set. Therefore, the set of transitions in the system is:

$$\begin{aligned} &\text{(A1)} \quad \textbf{true} \longrightarrow \langle \exists\, i \mid i \notin \text{theSet}_A \rhd \text{theSet}_A' = \text{theSet}_A \cup \{i\} \rangle \\ &\text{(A2)} \quad \textbf{true} \longrightarrow \textbf{stop} \end{aligned}$$

2. Transition (A1) is chosen; the integer 5 is added to $theSet_A$. The set of transitions in the system changes to the following:

$$\begin{aligned} &\text{(A1)} \quad \textbf{true} \longrightarrow \langle \exists\, i \mid i \notin \text{theSet}_A \rhd \text{theSet}_A' = \text{theSet}_A \cup \{i\} \rangle \\ &\text{(A2)} \quad \textbf{true} \longrightarrow \textbf{stop} \\ &\text{(A3}_5\text{)} \quad 5 \geq 731 \longrightarrow \text{p: } p' = \textbf{new } \text{ExampleSystemComponent} \end{aligned}$$

3. Transition ($A3_5$) is chosen. The precondition does not hold, so no state change occurs, and no change is made to the transition set.

4. Transition (A1) is chosen; the integer 1013 is added to theSet_A. The set of transitions in the system changes to the following:

$$\begin{aligned} &\text{(A1)} \quad \textbf{true} \longrightarrow \langle \exists\, i \mid i \notin \text{theSet}_A \rhd \text{theSet}_A' = \text{theSet}_A \cup \{i\} \rangle \\ &\text{(A2)} \quad \textbf{true} \longrightarrow \textbf{stop} \\ &\text{(A3}_5\text{)} \quad 5 \geq 731 \longrightarrow \text{p: } p' = \textbf{new } \text{ExampleSystemComponent} \\ &\text{(A3}_{1013}\text{)} \quad 1013 \geq 731 \longrightarrow \text{p: } p' = \textbf{new } \text{ExampleSystemComponent} \end{aligned}$$

5. Transition (A1) is chosen; the integer -7 is added to $theSet_A$. The set of transitions in the system changes to the following:

$$\begin{aligned}
\text{(A1)}\quad & \textbf{true} \longrightarrow \langle \exists\, i \mid i \notin \text{theSet}_A \rhd \text{theSet}_A' = \text{theSet}_A \cup \{i\} \rangle \\
\text{(A2)}\quad & \textbf{true} \longrightarrow \textbf{stop} \\
\text{(A3}_{-7}\text{)}\quad & -7 \geq 731 \longrightarrow p{:}\ p' = \textbf{new}\ \text{ExampleSystemComponent} \\
\text{(A3}_{5}\text{)}\quad & 5 \geq 731 \longrightarrow p{:}\ p' = \textbf{new}\ \text{ExampleSystemComponent} \\
\text{(A3}_{1013}\text{)}\quad & 1013 \geq 731 \longrightarrow p{:}\ p' = \textbf{new}\ \text{ExampleSystemComponent}
\end{aligned}$$

6. Transition ($A3_{1013}$) is chosen. The precondition holds, so a new process (to which we assign the identifier B) is created. B's state variable *theSet* is initialized to the empty set. Therefore, the set of transitions in the system changes to the following:

$$\begin{aligned}
\text{(A1)}\quad & \textbf{true} \longrightarrow \langle \exists\, i \mid i \notin \text{theSet}_A \rhd \text{theSet}_A' = \text{theSet}_A \cup \{i\} \rangle \\
\text{(A2)}\quad & \textbf{true} \longrightarrow \textbf{stop} \\
\text{(A3}_{-7}\text{)}\quad & -7 \geq 731 \longrightarrow p{:}\ p' = \textbf{new}\ \text{ExampleSystemComponent} \\
\text{(A3}_{5}\text{)}\quad & 5 \geq 731 \longrightarrow p{:}\ p' = \textbf{new}\ \text{ExampleSystemComponent} \\
\text{(A3}_{1013}\text{)}\quad & 1013 \geq 731 \longrightarrow p{:}\ p' = \textbf{new}\ \text{ExampleSystemComponent} \\
\text{(B1)}\quad & \textbf{true} \longrightarrow \langle \exists\, i \mid i \notin \text{theSet}_B \rhd \text{theSet}_B' = \text{theSet}_B \cup \{i\} \rangle \\
\text{(B2)}\quad & \textbf{true} \longrightarrow \textbf{stop}
\end{aligned}$$

7. Transition (B1) is chosen; the integer 731 is added to $theSet_B$. The set of transitions in the system changes to the following:

$$\begin{aligned}
\text{(A1)}\quad & \textbf{true} \longrightarrow \langle \exists\, i \mid i \notin \text{theSet}_A \rhd \text{theSet}_A' = \text{theSet}_A \cup \{i\} \rangle \\
\text{(A2)}\quad & \textbf{true} \longrightarrow \textbf{stop} \\
\text{(A3}_{-7}\text{)}\quad & -7 \geq 731 \longrightarrow p{:}\ p' = \textbf{new}\ \text{ExampleSystemComponent} \\
\text{(A3}_{5}\text{)}\quad & 5 \geq 731 \longrightarrow p{:}\ p' = \textbf{new}\ \text{ExampleSystemComponent} \\
\text{(A3}_{1013}\text{)}\quad & 1013 \geq 731 \longrightarrow p{:}\ p' = \textbf{new}\ \text{ExampleSystemComponent} \\
\text{(B1)}\quad & \textbf{true} \longrightarrow \langle \exists\, i \mid i \notin \text{theSet}_B \rhd \text{theSet}_B' = \text{theSet}_B \cup \{i\} \rangle \\
\text{(B2)}\quad & \textbf{true} \longrightarrow \textbf{stop} \\
\text{(B3}_{731}\text{)}\quad & 731 \geq 731 \longrightarrow p{:}\ p' = \textbf{new}\ \text{ExampleSystemComponent}
\end{aligned}$$

8. Transition (A2) is chosen, removing process A from the system. The set of transitions in the system changes to the following:

$$\begin{aligned}
\text{(B1)}\quad & \textbf{true} \longrightarrow \langle \exists\, i \mid i \notin \text{theSet}_B \rhd \text{theSet}_B' = \text{theSet}_B \cup \{i\} \rangle \\
\text{(B2)}\quad & \textbf{true} \longrightarrow \textbf{stop} \\
\text{(B3}_{731}\text{)}\quad & 731 \geq 731 \longrightarrow p{:}\ p' = \textbf{new}\ \text{ExampleSystemComponent}
\end{aligned}$$

9. Transition (B2) is chosen, removing process B from the system. The set of transitions in the system is now empty, so execution of the system terminates.

We note that, while this particular execution of this system terminates, not all executions of it do; there are many possible executions of this system that run forever.

2.3.2 Messaging

Dynamic UNITY's messaging system implements first-in first-out asynchronous messaging between every outbox/inbox pair. That is, all messages sent from a particular outbox to a particular inbox arrive in the order in which they were sent. The messaging system makes no guarantees about the ordering of messages sent to different inboxes from the same outbox, nor does it make any guarantees about the ordering of messages sent to the same inbox from different outboxes.

Semantically, inboxes and outboxes are treated as sequences of records, as discussed in Section 2.2.8.3. Each inbox (outbox) has a name, as well as two attributes that indicate how many messages have been placed into the inbox (outbox) and how many messages have been read (sent). Additionally, each message record in an inbox (outbox) has two (three) attributes that contain the message data and information about the message's source (destination), and each message record in an outbox has a flag indicating whether or not that message has been delivered to its destination inbox. For the purposes of our semantics, these attributes are updated atomically: when a message is delivered, the attributes of the source outbox and the destination outbox are all updated within the same atomic operation to reflect the delivery of that message.

Both inboxes and outboxes are empty when they are initially constructed (either at process initialization time or, for inboxes only, when instantiated during execution). All modifications of inbox and outbox contents and attributes are made either as part of normal transitions that use messaging operations during program execution, or by the messaging system itself. We now describe the semantics of the messaging operations.

2.3.2.1 Messaging Operations

As described in Section 2.2.8.3, outboxes support a single operation that allows for the sending of messages or sequences of messages, while inboxes support three operations that allow for the reception of messages and the detection of available messages. These operations are exactly equivalent to the following predicates, which use or modify the contents and attributes of mailboxes:

Send The **send** operation on an outbox is equivalent to an appropriate increment of that outbox's *len* attribute and a corresponding modification of the outbox's contents. Each process has a single outbox, which we represent by $\mathcal{O}$ for the purposes of this section. Since the **send** operation can take multiple forms, we give examples of each with their equivalent predicates:

Example 2.13 (Send operations)

1. A simple send operation—**send**(targetProcess, targetInboxName, sentMessage)—is equivalent to the following predicate:

   $\mathcal{O}'[\mathcal{O}.\textbf{len}] = \{\text{targetProcess, targetInboxName, sentMessage}\} \wedge$
   $\mathcal{O}'.\textbf{len} = \mathcal{O}.\textbf{len} + 1$

2. A multiple send operation—**send**(targetProc1, targetInbox1, sentMessage1, targetProc2, targetInbox2, sentMessage2)—is equivalent to the following predicate:

   $\mathcal{O}'[\mathcal{O}.\textbf{len}] = \{\text{targetProc1, targetInbox1, sentMessage1}\} \wedge$
   $\mathcal{O}'[\mathcal{O}.\textbf{len} + 1] = \{\text{targetProc2, targetInbox2, sentMessage2}\} \wedge$
   $\mathcal{O}'.\textbf{len} = \mathcal{O}.\textbf{len} + 2$

3. A quantified send operation—**send**($\langle$, p | p $\in$ targetProcesses $\triangleright$ p, "inboxName", MessageType(p)$\rangle$)—is equivalent to the following predicate (recall that the quantification of a quantified message send must be finite):

   $\langle \forall$ p | p $\in$ targetProcesses $\triangleright$
   $\quad \langle \exists$ i | $\mathcal{O}.\textbf{len} \leq$ i $< \mathcal{O}'.\textbf{len} \triangleright \mathcal{O}'[i] = \{$p, "inboxName", MessageType(p)$\}\rangle \wedge$
   $\mathcal{O}'.\textbf{len} = \mathcal{O}.\textbf{len} +$ |targetProcesses|

Probe The operation *inbox-var*.**probe** is equivalent to the predicate *inbox-var*.**cnt** < *inbox-var*.**len**, which compares the *cnt* and *len* attributes of an inbox. If the predicate evaluates to **true**, then there is a message on the inbox that has not yet been read.

Current The operation *inbox-var*.**current** is equivalent to the term *inbox-var*[*inbox-var*.**cnt**]; it is the message record contained at the index of the inbox corresponding to the number of messages which have already been read. Typically, the current operation is used to access one or more fields of the message record, as in *inbox-var*.**current**.msg (to retrieve the actual message content).

Advance The operation *inbox-var*.**advance** is equivalent to the predicate *inbox-var′*.**cnt** = *inbox-var*.**cnt** + 1, which increments by 1 the number of messages which have been read from the inbox. This equivalence holds regardless of the value of the inbox's *len* attribute, which means that it is possible for *cnt* to exceed *len*. This is acceptable because we have defined inboxes as infinite sequences, although it will result in default data being read from the inbox until enough messages are placed in the inbox for *len* to catch up to *cnt*. To avoid this (usually undesirable) behavior, advance should be used on an inbox only when a probe of the inbox evaluates to **true**.

Chapter 3

Verification of Dynamic UNITY Specifications

In this chapter, we introduce and discuss a logic for the verification of Dynamic UNITY specifications. Verification of individual programs in a Dynamic UNITY system is very similar to verification of UNITY programs using UNITY logic [8], while verification of Dynamic UNITY systems in their entirety is accomplished by first verifying properties of the systems' component programs and then reasoning about their messaging interactions.

We first present some notation and basic concepts that we will use throughout this chapter and during our proofs in the example chapters; then, we discuss the specific proof rules and theorems that provide a basis for proving properties of Dynamic UNITY programs and systems.

3.1 Basic Concepts and Conventions

3.1.1 Quantification

We use various types of quantification while stating the proof rules and theorems of this chapter, as well as in our proofs of Dynamic UNITY specifications. Our notation for quantification is taken from Leino [37]. In general, a quantification is $\langle op\ boundvars \mid ranges \triangleright expression \rangle$, where *op* is the operator of the quantification, *boundvars* is the set of bound variables, *ranges* is a predicate restricting the ranges of the bound variables, and *expression* is the expression to be quantified. The operator of the quantifier must be associative and commutative, and must have an identity element. A variable with no range specification is quantified over the entire range of its type; if there are no range specifications in the quantifier, the shorter $\langle op\ boundvars \triangleright expression \rangle$ form can be used. If an empty range is specified (there are no values for the bound variables which satisfy the range predicate), the value of the quantifier is the identity element of *op*.

EXAMPLE 3.1 (QUANTIFICATIONS)

1. Universal: $\langle \forall i \mid 0 \leq i \leq N \rhd a[i] = 0 \rangle$ holds if every element of array a with an index between 0 and N inclusive equals 0, and does not hold otherwise. If the range of a universal quantification is empty, the quantification holds. By convention, we use the $\forall$ symbol instead of the $\wedge$ operator in universal quantifications.

2. Existential: $\langle \exists i \mid 0 \leq i \leq N \rhd a[i] = 0 \rangle$ holds if there is at least one element of array a with an index between 0 and N inclusive that equals 0, and does not hold otherwise. If the range of an existential quantification is empty, the quantification does not hold. By convention, we use the $\exists$ symbol instead of the $\vee$ operator in existential quantifications.

3. Summation: $\langle \Sigma i \mid 0 \leq i \leq N \rhd a[i] \rangle$ evaluates to the sum of all elements of array a with indices between 0 and N inclusive. If the range of a summation is empty, the summation evaluates to 0. By convention, we use the Σ symbol instead of the $+$ operator in summations.

The everywhere operator, denoted by enclosing a predicate in square brackets ([]), is a special instance of quantification. Introduced by Dijkstra and Scholten [17] as a function from predicates to Boolean values, it is an implicit universal quantification of the enclosed predicate over all states. We use the everywhere operator in some of the definitions and theorems of this chapter.

3.1.2 Assertions

In Hoare logic [25, 26], the assertion $\{p\}\ s\ \{q\}$ (usually referred to as a Hoare triple) denotes that execution of statement s in any state that satisfies predicate p either terminates in a state that satisfies predicate q or does not terminate. We use a similar construct to make statements about the execution of Dynamic UNITY transitions: if the assertion $\{p\}\ g \longrightarrow t\ \{q\}$ holds, it means that the execution of transition $g \longrightarrow t$ in any state that satisfies p either terminates in a state that satisfies q or does not terminate. Since all well-formed Dynamic UNITY transitions terminate (by definition), we will avoid the issue of nonterminating computations by assuming for the remainder of this discussion that all transitions are well-formed.

The execution of the transition $g \longrightarrow t$, as discussed previously, either establishes t (if g holds) or is equivalent to **skip** (if g doesn't hold). Thus, our $g \longrightarrow t$ is equivalent to the ***if*** $B \longrightarrow S\ [] \neg B \longrightarrow$ *skip* ***fi*** conditional described by Dijkstra and Scholten [17] and by Morgan [52], where B is the guard g and S is a statement that establishes the binary predicate t on the pre- and post-states.

We now formally define the semantics of a Dynamic UNITY transition. In these definitions,

suppose S is a statement that establishes the binary predicate t on the pre- and post-states. In terms of Hoare triples, a transition is defined as follows:

$$\{p\}\ g \longrightarrow t\ \{q\} \triangleq \{p \wedge g\}\ S\ \{q\} \wedge \{p \wedge \neg g\}\ skip\ \{q\} \tag{3.1}$$

In terms of weakest preconditions, a transition is defined as follows (instantiating the Dijkstra/Scholten definition of ***if***):

$$[wp.(g \longrightarrow t).X \triangleq (g \Rightarrow wp.S.X) \wedge (\neg g \Rightarrow wp.skip.X)] \tag{3.2}$$

Thus, in order to prove $\{p\}\ g \longrightarrow t\ \{q\}$, we must show the following (where q' is q with all its free variables primed; that is, q in the post-state):

$$(p \wedge g \wedge t \Rightarrow q') \wedge (p \wedge \neg g \Rightarrow q) \tag{3.3}$$

Assertions may be quantified using any quantifier that operates on the Boolean values. We sometimes write assertions as $\{p\}\ s\ \{q\}$, where s is known to be a Dynamic UNITY transition (as in a quantification over the minimal set of transitions in a running process). The context will always eliminate any possible confusion between one of our assertions and a traditional Hoare triple.

3.1.3 Functions and Operators

We use all the usual mathematical and logical operators, as well as some temporal operators that will be defined later in this chapter. We denote function application, including the application of unary operators on predicates (such as **transient**), with the "." operator. We also denote operation invocation and access to data structures with the "." operator, but the usage of the operator is always clear from context. Function application always associates to the left. We adopt the following conventions regarding the precedence of logical relations. All relations in each group have the same precedence, and the groups are listed in order of increasing precedence:

1. **follows** , $\leadsto$ (**leads-to**), **next**
2. $\equiv$
3. $\Rightarrow$, $\Leftarrow$
4. $\wedge$, $\vee$
5. $=$, $\neq$

6. mathematical operators, with their usual binding powers
7. $\neg$
8. . (function application, operation execution), $\downarrow_i$ (filtering operator), $\bowtie$ (concatenation operator)

For any two predicates p and q, $p \equiv q$ and $p = q$ have identical meanings. However, since the precedences of $\equiv$ and $=$ differ, we can unambiguously write expressions such as $p = r \equiv q = s$. Note that the $\equiv$ operator is only defined on predicates, while the $=$ operator is defined on everything.

The filtering operator ($\downarrow_k$), where k is a bound variable of the operator, is used primarily to examine subsequences of message histories but can also be used for other purposes. It is a binary operator taking a set, sequence or array as its first operand and a predicate as its second; it applies the predicate to every element in the first operand (denoted by k in the predicate) and returns a set, sequence, or array (respectively) containing only the elements of the first operand that satisfy the predicate with their ordering (if any) preserved. For example, if x is the integer sequence $\langle 1, 1, 2, 3, 1, 4, 5, 7\rangle$, then $x \downarrow_k (k > 1)$ evaluates to the sequence $\langle 2, 3, 4, 5, 7\rangle$.

The concatenation operator ($\bowtie$) is used to generate sequences from other sequences or elements. It takes two operands, each a sequence or an element, and returns the sequence generated by concatenating them. For example, if x is the integer 3, and y is the integer sequence $\langle 4, 5, 6\rangle$, $x \bowtie y$ evaluates to the sequence $\langle 3, 4, 5, 6\rangle$.

3.2 Formal Execution Model

In this section, we formalize the execution model described in Section 2.3.1. We first formalize the execution model for programs, and then the execution model for systems. These models are constructed in such a way that it is possible to prove properties of Dynamic UNITY programs that remain applicable when those programs are incorporated into larger systems. This forms the basis for reasoning about multiple communicating processes in Dynamic UNITY.

3.2.1 Program Executions

Each execution of a Dynamic UNITY program P is an infinite sequence of pairs (S_i, T_i) for $i \geq 0$, where S_i is the execution state at step i and T_i is the transition of P to be executed at step i. Every nontrivial program has an infinite number of possible executions. We denote the set containing all possible executions of a particular program P by $P.\mathcal{X}$, and the subset of $P.\mathcal{X}$ containing only executions where the program's instantiation parameters satisfy a predicate p

by $P.\mathcal{X}(p)$. There is a special execution state, called the uninitialized state and denoted by $\mathcal{U}$, which holds before the initialization of the execution state, and a special transition, denoted by **initialize**, which takes the system from the uninitialized state to an initial state.

The strongest predicate that holds on a state S—that is, the predicate that completely specifies the values of all variables in the state S—is denoted by $\text{Pred}(S)$. T_i may be **skip**, since all programs implicitly contain **skip** as a transition. We sometimes denote T_i explicitly as a guarded transition $g_i \longrightarrow t_i$.

The transition set of a Dynamic UNITY program changes with the execution state, due to the existence of quantified transitions. Given a complete specification of a program's state, it is always possible to determine the exact transition set for the program from the program text. However, in order to prove program properties, we will often need to determine the transition set using an incomplete specification of the program state. Given any predicate constraining the program state, we define minimal and maximal transition sets as follows.

Definition 3.1 (Minimal Transition Set) *The* minimal transition set *of a program* P *constrained by a predicate* p *contains exactly those fair transitions of* P *that exist in all program states* S *for which* $\text{Pred}(\text{S}) \Rightarrow \text{p}$. *That is, the minimal transition set is the intersection of the fair transition sets for all program states satisfying* p. *We denote the minimal transition set of* P *constrained by* p *as* $\text{P}.\mathcal{T}^{-}(\text{p})$.

Definition 3.2 (Maximal Transition Set) *The* maximal transition set *of a program* P *constrained by a predicate* p *contains exactly those (fair and unfair) transitions of* P *that exist in any program state* S *for which* $\text{Pred}(\text{S}) \Rightarrow \text{p}$. *That is, the maximal transition set is the union of the transition sets for all program states satisfying* p. *We denote the maximal transition set of* P *constrained by* p *as* $P.\mathcal{T}^{+}(\text{p})$.

We sometimes denote the transition sets of a particular program execution using that execution's label instead of its program's; i.e., $R.\mathcal{T}^{-}(p)$ instead of $P.\mathcal{T}^{-}(p)$, where R is an execution of P.

Example 3.2 (Transition sets)

The minimal and maximal transition sets of a simple Dynamic UNITY program (Specification 3.1) constrained by various predicates are listed in Table 3.1. We denote instances of transition (2) with $i = k$ by 2_k.

Every program execution state contains the outbox state for the single outbox of the execution, the inbox state for every inbox created up to that point in the execution, and the values of the conventional state variables of the program. We denote the outbox state at step i by

program ExampleProgram

declare
theSet: **set** {**integer**}

initially
theSet = ∅

fair-transition
(1) **true** ⟶ ⟨∃ i | i ∉ theSet ▷ theSet′ = theSet ∪ {i}⟩
(2) [] ⟨[] i | i ∈ theSet ▷ i ≥ 1138 ⟶ p: p′ = **new** ExampleSystemComponent⟩

unfair-transition
(3) **true** ⟶ theSet′ = ∅

end

Specification 3.1: An example program used to illustrate minimal and maximal transition sets

p	ExampleProgram.$\mathcal{T}^-$(p)	ExampleProgram.$\mathcal{T}^+$(p)
true	$\{1\}$	$\{1, 2_0, 2_{-1}, 2_1, 2_{-2}, 2_2, \ldots, 3\}$
$theSet = \emptyset$	$\{1\}$	$\{1, 3\}$
$\langle \forall i \mid i \in theSet \triangleright 0 \leq i \leq 3 \rangle$	$\{1\}$	$\{1, 2_0, 2_1, 2_2, 2_3, 3\}$
$\{11, 21\} \subseteq theSet$	$\{1, 2_{11}, 2_{21}\}$	$\{1, 2_0, 2_{-1}, 2_1, 2_{-2}, 2_2, \ldots, 3\}$
$theSet = \{1, 27, 36\}$	$\{1, 2_1, 2_{27}, 2_{36}\}$	$\{1, 2_1, 2_{27}, 2_{36}, 3\}$

Table 3.1: Transition sets corresponding to various predicates on the state of the program in Specification 3.1.

$\mathcal{O}_i$, the outbox state as a variable (for use in theorems and proofs) by $\mathcal{O}$, the set containing the names of all inboxes existing at step i by $\mathcal{I}_i$, the state of the inbox with name k at step i by $\mathcal{I}_i^k$, and the inbox with name k as a variable (for the purpose of quantifying over inbox variables in theorems and proofs) by $\mathcal{I}^k$. Every execution state S_i consists of a volatile and a non-volatile portion, which are defined as follows.

Definition 3.3 (Volatile and Non-volatile State) *The* volatile *portion of an execution state* S_i *consists of the* sent *fields for the messages in* $\mathcal{O}_i$ *and* $\mathcal{I}_i^k$.history *(the sequence of messages which have been delivered to inbox* k*) for all inboxes* k *in* $\mathcal{I}_i$. *The* non-volatile *portion of an execution state* S_i *consists of everything else in* S_i.

The volatile portion of an execution state can only be modified by the messaging system, while the non-volatile portion can only be modified by program transitions. The volatile and non-volatile portions of the state are non-intersecting, and are denoted (respectively) by $\mathcal{V}.S_i$ and $\overline{\mathcal{V}}.S_i$.

An individual program execution is also called a *process*. Every process has a globally unique label associated with it; when referring to the states and transitions of the process with label

l, we use a superscript l (as in S_i^l, T_i^l) to denote the process we are referring to. Processes are subject to certain safety and progress constraints, which we describe in the following sections.

3.2.1.1 Safety Constraints

The following constraints on processes are safety constraints. They restrict the possible states and transitions for each step of an execution. We denote the globally unique label of the process under discussion by L.

Initial Step Every execution has an *initial step*, hereafter denoted by I, such that I is the earliest step for which the execution state is not uninitialized (recall that the uninitialized state is denoted by $\mathcal{U}$).

$$I = \langle \min i \mid 0 \leq i \rhd S_i \neq \mathcal{U} \rangle \tag{3.4}$$

Pre-Initial Transitions The transition at the execution step before the initial step is **initialize**, and the transition at every step before the **initialize** transition is **skip**. Note that if the initial step is step 0, the **initialize** transition implicitly occurs before step 0 (at "step -1").

$$T_{I-1} = \textbf{initialize} \wedge \langle \forall i \mid 0 \leq i < I - 1 \rhd T_i = \textbf{skip} \rangle \tag{3.5}$$

Initial State The *initial state*, S_I, is specified by the parameters and the initially-section of the program. This state need not be the same for all executions, since there may be variation in the initializations of variables due to differing program parameters or nondeterministic initializations.

Transitions and Subsequent States Each transition T_i, $i \geq I$, is chosen from the maximal transition set corresponding to S_i (Equation 3.6), and the change in the nonvolatile state between S_i and S_{i+1} results from the execution of transition T_i (Equation 3.7):

$$\langle \forall i \mid I \leq i \rhd T_i \in R.\mathcal{T}^+(\text{Pred}(S_i)) \rangle \tag{3.6}$$

$$\langle \forall i \mid I \leq i \rhd \{\text{Pred}(\overline{V}.S_i)\}\ g_i \longrightarrow t_i\ \{\text{Pred}(\overline{V}.S_{i+1})\} \rangle \tag{3.7}$$

Messaging Safety The set of inboxes in the execution is monotonically nondecreasing (Equation 3.8); for every inbox, the message sequence and the index of the current message are monotonically nondecreasing (Equation 3.9); the message sequence and message delivery states in the outbox are monotonically nondecreasing (Equation 3.10); and for every inbox in the execution state at a particular execution step, the sequence of messages in that inbox from process

L is the same as the sequence of delivered messages in process L's outbox addressed to that inbox (Equation 3.11). We denote the sequence consisting of only the *msg* fields of the elements of sequence *history* by *history.msg*. For inboxes i, j, the equation $i \preceq j$ means that the **name** attributes of i and j are equal, the **cnt** and **len** attributes of i are less than or equal to the corresponding attributes of j, and the sequences of *msg* and *proc* fields in i are subsequences of the corresponding sequences in j. For outboxes i, j, the equation $i \preceq j$ means that the **len** attribute of i is less than or equal to the **len** attribute of j, that the sequences of *msg*, *proc* and *mbox* fields in i are initial subsequences of the corresponding sequences in j, and that for every *del* field of i which has the value **true**, the *del* field of j also has the value **true**.

$$\langle \forall i \mid 0 \le i \rhd \mathcal{I}_i \subseteq \mathcal{I}_{i+1} \rangle \tag{3.8}$$

$$\langle \forall i, b \mid 0 \le i \wedge b \in \mathcal{I}_i \rhd \mathcal{I}_i^b \preceq \mathcal{I}_{i+1}^b \rangle \tag{3.9}$$

$$\langle \forall i \mid 0 \le i \rhd \mathcal{O}_i \preceq \mathcal{O}_{i+1} \rangle \tag{3.10}$$

$$\begin{aligned} \langle \forall i, b \mid 0 \le i \wedge b \in \mathcal{I}_i \rhd \ & (\mathcal{I}_i^b \downarrow_m (m.\text{proc} = L)).\text{msg} = \\ & \mathcal{O}_i \downarrow_m (m.\text{del} \wedge m.\text{proc} = L \wedge m.\text{mbox} = b).\text{msg} \rangle \end{aligned} \tag{3.11}$$

Termination If there exists an i, $0 \le i$, such that T_i contains a **stop** command that is executed, then for all $j > i$, T_j = **skip**. That is, once a **stop** command is executed, the nonvolatile state of the process stops changing.

3.2.1.2 Progress Constraints

The following constraints on program executions are progress constraints. They ensure that weak fairness is guaranteed for transition executions and that all sent messages are eventually delivered.

Weak Fairness As defined in Chapter 2, weak fairness means that every fair transition is infinitely often either selected or not available for execution. This is equivalent to saying that if a transition is available for execution at every step after a certain step k, and is guaranteed to remain in the transition set at least until it is executed, then the transition is executed at some step after k. This is expressed by the following equation.

$$\langle \forall s, k \mid k \ge 0 \wedge s \in P.\mathcal{T}^{-}(S_k) \rhd \langle \exists j \mid j \ge k \rhd (T_j = s) \vee (s \notin P.\mathcal{T}^{-}(S_j)) \rangle \rangle \tag{3.12}$$

Message Delivery In order to reason about message communication, we need the constraint that all sent messages are eventually delivered to their destinations. Within the process labelled L, this is expressed by the following equation.

$$\langle \forall i, b, k \mid i \geq 0 \wedge b \in \mathcal{I}_i \wedge k \preceq (\mathcal{O}_i \downarrow_m (m.\text{proc} = L \wedge m.\text{mbox} = b)).\text{msg} \rhd \langle \exists j \mid j > i \rhd k \preceq (\mathcal{I}_j^b \downarrow_m (m.\text{proc} = L)).\text{msg} \rangle \rangle \quad (3.13)$$

3.2.2 System Executions

Each execution of a Dynamic UNITY system consists of an infinite sequence of tuples (S_i, T_i, L_i, Z_i), where S_i is the execution state of the system at step i, T_i is the transition of X to be executed at step i, L_i is a set containing the labels of all running processes in the system at step i, and Z_i is a set containing the labels of all stopped processes in the system at step i. As in program executions, T_i may be **skip**. Every nontrivial system has an infinite number of possible executions; we denote the set containing all possible executions for a particular system X by $X.\mathcal{X}$.

We use notation similar to that for program executions (for example, the maximal transition set for a system X constrained by a predicate p is $X.\mathcal{T}^+(p)$). In particular, to denote the set containing the (process-qualified) names of all inboxes in the system at step i we write $\mathcal{I}_i$, to denote the set containing the names of all inboxes belonging to a particular process l at step i we write $\mathcal{I}_i^l$, to explicitly denote the inbox named "in" belonging to a particular process l at step i we write $\mathcal{I}_i^{l.\text{in}}$, to denote the inbox named "in" belonging to a particular process l as a variable we write $\mathcal{I}^{l.\text{in}}$, to denote the outbox state of process l at step i we write $\mathcal{O}_i^l$, and to denote the outbox state of process l as a variable we write $\mathcal{O}^l$. System executions are subject to certain safety and progress constraints, which we describe in the following sections.

3.2.2.1 Safety Constraints

The following constraints on system executions are safety constraints. They restrict the possible states and transitions for each step of an execution.

Initial Processes The set of running processes at the initial step, L_0, consists of an instantiation of the system's initial program and any processes created as a result of that program's initialization; these are called the *initial processes*. The set of stopped processes at the initial step, Z_0, is empty because it is not possible for any process to execute a **stop** statement before the initial step.

Initial State The *initial state*, S_0, is specified by the parameters and the initially-section of the system's initial program and any processes created as a result of that program's initialization. This state need not be the same for all executions, since there may be variation in the initialization of variables due to differing program parameters or nondeterministic initializations.

Transitions and Subsequent States Each transition T_i, $i \geq 0$, is chosen from the transition set corresponding to S_i (Equation 3.14), and the change in the nonvolatile state between S_i and S_{i+1} results from the execution of transition T_i (Equation 3.15):

$$\langle \forall i \mid 0 \leq i \rhd T_i \in X.\mathcal{T}(S_i) \rangle \tag{3.14}$$

$$\langle \forall i \mid 0 \leq i \rhd \{\overline{V}.S_i\}\ g_i \longrightarrow t_i\ \{\overline{V}.S_{i+1}\} \rangle \tag{3.15}$$

Process Set Monotonicity The set of processes (both running and stopped) in the system is monotonically nondecreasing (Equation 3.16); the set of stopped processes in the system is monotonically nondecreasing (Equation 3.17); the intersection of the sets of running and stopped processes is empty (Equation 3.18).

$$\langle \forall i \mid 0 \leq i \rhd (L_i \cup Z_i \subseteq L_{i+1} \cup Z_{i+1}) \rangle \tag{3.16}$$

$$\langle \forall i \mid 0 \leq i \rhd Z_i \subseteq Z_{i+1} \rangle \tag{3.17}$$

$$\langle \forall i \mid 0 \leq i \rhd L_i \cap Z_i = \varnothing \rangle \tag{3.18}$$

System State and Process States The system state at every execution step is the Cartesian product of the states of the processes in the system at that step.

$$\langle \forall i \mid 0 \leq i \rhd S_i = \langle \times l \mid l \in L_i \rhd S_i^l \rangle \rangle \tag{3.19}$$

Execution Noninterference Every execution step of the system can be expressed as an execution step of one running process in the system (Equations 3.20, 3.21). If one running process has a non-**skip** transition, it is executed and the nonvolatile portion of that process's state is changed, while the nonvolatile portions of other processes' states are not changed (Equation 3.22).

$$\langle \forall i \mid 0 \leq i \rhd \langle \exists l \mid l \in L_i \rhd T_i = T_i^l \rangle \wedge \langle \forall l \mid l \in L_i \wedge T_i \neq T_i^l \rhd T_i^l = \textbf{skip} \rangle \rangle \tag{3.20}$$

$$\langle \forall i, l \mid 0 \leq i \wedge l \in L_i \wedge T_i = T_i^l \rhd \{\overline{V}.S_i^l\}\ g_i \longrightarrow t_i\ \{\overline{V}.S_{i+1}^l\} \rangle \tag{3.21}$$

$$\langle \forall i, l \mid 0 \leq i \wedge l \in L_i \wedge T_i \neq T_i^l \rhd \overline{V}.S_i^l = \overline{V}.S_{i+1}^l \rangle \tag{3.22}$$

Process Set Determinism For all i, $0 \leq i$, if process l is in L_{i+1} and not in L_i, then process l must be created as a result of transition T_i. For all i, $0 \leq i$, if process l is stopped at step $i+1$, i.e., l is in Z_{i+1}, then either (a) process l was already stopped at step i (l is in Z_i, and T_i^l is **skip**) or (b) process l stops at step i (l is in L_i and T_i^l contains a **stop**). That is, processes cannot appear in the system after the initial step without having been created by other processes and cannot be stopped except by transitions of their own, and stopped processes can have no non-skip transitions. Therefore, stopped processes cannot re-enter the system.

Messaging Safety The messaging safety rules specified for processes in Equations 3.8–3.11 hold for systems as well. In addition, for every inbox and process in the execution state at a particular execution step, the sequence of messages in that inbox delivered from that process is the same as the sequence of delivered messages in that process's outbox addressed to that inbox.

$$\begin{aligned}&\langle \forall p, q, i, b \mid p \in L_i \cup Z_i \wedge q \in L_i \cup Z_i \wedge 0 \leq i \wedge b \in \mathcal{I}_{i+1}^q \rhd \\ &\quad (\mathcal{I}_{i+1}^{q.b} \downarrow_m (m.\text{proc} = p)).\text{msg} = (\mathcal{O}_i^p \downarrow_m (m.\text{del} \wedge m.\text{proc} = q \wedge m.\text{mbox} = b)).\text{msg}\rangle \end{aligned} \quad (3.23)$$

3.2.2.2 Progress Constraints

The following constraints on system executions are progress constraints. They ensure that weak fairness is guaranteed for transition executions and that all sent messages are eventually delivered.

Weak Fairness If a transition is available for execution at every step after a certain step k and is guaranteed to remain in the transition set at least until it is executed, then the transition is executed at some step after k. This is expressed by the following equation.

$$\langle \forall s, k \mid k \geq 0 \wedge s \in X.\mathcal{T}^-(S_k) \rhd \langle \exists j \mid j \geq k \rhd T_j = r \vee r \notin X.\mathcal{T}^-(S_j)\rangle\rangle \quad (3.24)$$

Message Delivery In order to reason about message communication, we need the constraint that all sent messages are eventually delivered to their destinations. Within a system execution, this is expressed by the following equation.

$$\begin{aligned}&\langle \forall p, q, i, b, k \mid p \in L_i \cup Z_i \wedge q \in L_i \cup Z_i \wedge 0 \leq i \wedge b \in \mathcal{I}_i^q \wedge \\ &\qquad k \preceq (\mathcal{O}_i^p \downarrow_m (m.\text{proc} = q \wedge m.\text{mbox} = b)).\text{msg} \rhd \\ &\qquad\qquad \langle \exists j \mid j > i \rhd k \preceq (\mathcal{I}_j^{q.b} \downarrow_m (m.\text{proc} = L)).\text{msg}\rangle\rangle \end{aligned} \quad (3.25)$$

3.2.3 Subsystem Executions

A *subsystem* consists of a fixed subset of the processes of a system. A subsystem execution is the projection of a system execution which contains the states and transitions of the processes in the subsystem. Subsystem executions have the same form as system executions—infinite sequences of tuples $(S_i^V, T_i^V, L_i^V, Z_i^V)$, where V is the set of processes in the subsystem. The tuples which make up a subsystem execution are computed from the tuples of the corresponding system execution as follows:

- For all $i \geq 0$, $S_i^V = \langle \times l \mid l \in V \wedge l \in L_i \cup Z_i \rhd S_i^l \rangle$.
- For every transition T_i such that $T_i = T_i^l$, $l \in V$, $T_i^V = T_i$. For all other transitions T_i, $T_i^V =$ **skip**.
- For all $i \geq 0$, $L_i^V = L_i \cap V$ and $Z_i^V = Z_i \cap V$.

All our reasoning will be done on subsystems. We use notation for subsystems analogous to that for systems (so that, for example, the maximal transition set of a subsystem V constrained by predicate p is $V.\mathcal{T}^+(p)$). Typically, V will consist of either a single process l (and the subsystem will be the process l in isolation) or the entire set of processes for a system (and the subsystem will be the entire system). The benefit of this approach is that if we can prove properties about all executions of a particular program, these properties hold for all systems containing any execution of that program. We use the execution model to formalize our operational understanding of Dynamic UNITY executions. However, when proving properties of programs and systems, we try to use assertions rather than directly reasoning about subsystem execution sequences.

3.3 Fundamental Operators

In this section, we define the three fundamental operators **initially**, **next** and **transient** in terms of our execution model, and present proof rules for using them in our logical framework. In subsequent sections, we will use these fundamental operators to define other useful operators (**stable**, **invariant**, $\leadsto$ (**leads-to**), **follows**). These operators will form the basis for reasoning about our programs and systems. When these operators are used in proofs, the systems to which they apply will generally be understood from context. However, we explicitly specify them in the definitions and theorems of this chapter.

3.3.1 Initially

The **initially** operator allows us to formalize initial conditions for the execution of a subsystem. It is defined as follows:

Definition 3.4 (Initially Operator) *Given predicate* p *and subsystem* X: **initially**.p.X *holds if, for every possible execution of* X, p *holds in the initial state.*

$$\textbf{initially}.p.X \triangleq \langle \forall Y \mid Y \in X.\mathcal{X} \rhd [\mathrm{Pred}(Y.S_0) \Rightarrow p] \rangle$$

initially.p.X can be proven from the program texts which comprise subsystem X, because the initial condition of the system can be calculated directly from the texts of the declare and initially sections of the programs and the parameters passed to the initial programs.

3.3.2 Next

The **next** operator allows us to prove safety properties of a subsystem, by specifying restrictions on the next subsystem state given the current subsystem state. It is defined as follows:

Definition 3.5 (Next Operator) *Given predicates* p *and* q *and subsystem* X: (p **next** q).X *holds if, for every execution of* X*, every state in which* p *holds is immediately followed by a state in which* q *holds.*

$$(p \textbf{ next } q).X \triangleq \langle \forall Y, i \mid Y \in X.\mathcal{X} \wedge i \geq 0 \rhd [\mathrm{Pred}(Y.S_i) \Rightarrow p] \Rightarrow [\mathrm{Pred}(Y.S_{i+1}) \Rightarrow q] \rangle$$

Since stuttering steps are allowed in our execution model, this definition constrains p and q such that:

$$(p \textbf{ next } q).X \Rightarrow [p \Rightarrow q] \tag{3.26}$$

Calculating $(p$ **next** $q).X$ by directly using this definition is impractical, since it requires us to verify an infinite number of implications for an infinite number of execution sequences. However, we can use the knowledge that Dynamic UNITY systems instantiate their processes from a static set of programs and that the non-volatile state of a process can be manipulated only by that process's transitions to calculate stronger properties that imply $(p$ **next** $q).X$. The following are the resulting proof rules.

Proof Rule 1 (Next Property for a Set of Processes)
Given predicates p *and* q *over the non-volatile state of a set of processes* V *in subsystem* X: *if every transition in the maximal transition set of* V *constrained by* p *terminates in a state satisfying* q

when executed in a state satisfying p*, then* (p **next** q)*.X holds.*

$$\frac{\langle \forall s \mid s \in V.\mathcal{T}^+(p) \rhd \{p\}\ s\ \{q\} \rangle}{(p\ \textbf{next}\ q).X}$$

This proof rule is consistent with the definition of the **next** operator, because any transition of the system other than those in $V.\mathcal{T}^+(p)$ has an effect on V's non-volatile state equivalent to **skip**. We quantify over the maximal transition set to ensure that we take into account all transitions that can possibly be enabled when p holds. We can use this rule to prove next properties for a single process R, by making V the set containing only process R.

If p and q are predicates over volatile state (or a combination of volatile and non-volatile state), the semantics of the messaging system must be taken into account. This is typically done using the **follows** operator, discussed later in this chapter.

The **next** operator has previously appeared in the context of static systems in various forms, including the **co** operator in Misra [47], the *next* operator in Chandy and Sanders [9, 10], the **next** operator in Sivilotti [66], and the $\bigcirc$ operator in temporal logic [62].

3.3.3 Transient

The **transient** operator allows us to prove progress properties of a subsystem, by specifying that certain predicates on the state of a subsystem cannot hold forever.

Definition 3.6 (Transient Operator) *Given predicate* p *and subsystem* X: **transient**.p.X *holds if, for every execution of* X*, every state is followed by some later state in which* ¬p *holds.*

$$\textbf{transient}.p.X \triangleq \langle \forall Y, i \mid Y \in X.\mathcal{X} \wedge i \geq 0 \rhd \langle \exists j \mid j > i \rhd [\mathrm{Pred}(Y.S_j) \Rightarrow \neg p] \rangle \rangle$$

Calculating **transient**.p by directly using this definition is impractical, since it requires us to verify an infinite number of implications for an infinite number of execution sequences. However, we can use the knowledge that Dynamic UNITY systems instantiate their processes from a static set of programs and that the non-volatile state of a process can be manipulated only by that process's transitions to calculate stronger properties that imply **transient**.$p.X$. The following are the resulting proof rules.

Proof Rule 2 (Transient Operator for a Set of Processes)
Given a predicate p *over the non-volatile state of a set of processes* V *in subsystem* X*: if any transition in the minimal transition set of* V *constrained by* p *terminates in a state satisfying* ¬p

when executed in a state satisfying p*, then* **transient**.p.X *holds.*

$$\frac{\langle \exists s \mid s \in V.\mathcal{T}^{-}(p) \rhd \{p\}\ s\ \{\neg p\}\rangle}{\textbf{transient}.p.X}$$

This proof rule is consistent with the definition of the **transient** operator, because all transitions in $V.\mathcal{T}^{-}(p)$ are guaranteed to remain eligible for execution as long as p holds. Weak fairness guarantees that if one or more transitions in $V.\mathcal{T}^{-}(p)$ satisfy the assertion, either one such transition will execute eventually or p will be falsified by some transition outside of $V.\mathcal{T}^{-}(p)$. We can use this rule to prove transient properties for a single process R, by making V the set containing only process R.

If p is a predicate over volatile state (or a combination of volatile and non-volatile state), the semantics of the messaging system must be taken into account.

The **transient** operator has previously appeared in the context of static systems in Misra [46] and Sivilotti [66].

3.4 Derived Operators

In this section, we define some useful operators in terms of the fundamental operators defined in the previous section.

3.4.1 Stable

The **stable** operator allows us to state that, once a particular predicate on the state of a subsystem holds, it continues to hold thereafter.

Definition 3.7 (Stable Operator) *Given predicate* p *and subsystem* X: **stable**.p.X *holds if, for every execution of* X*, every state in which* p *holds is immediately followed by a state in which* p *holds. That is,* p *is never falsified once it has been established.*

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

3.4.2 Invariant

The **invariant** operator allows us to state that a particular predicate holds on every reachable state of a subsystem.

DEFINITION 3.8 (INVARIANT OPERATOR) *Given predicate* p *and subsystem* X: **invariant**.p.X *holds if, for every execution of* X, p *holds in every state.*

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

3.4.3 Leads-To

The $\leadsto$ (**leads-to**) operator is the primary operator used in progress proofs. It allows us to show that if a particular predicate on the state of a subsystem holds at some point during any execution, another predicate is guaranteed to hold at some later point.

DEFINITION 3.9 (LEADS-TO OPERATOR) *Given predicates* p *and* q *and subsystem* X: (p $\leadsto$ q).X *holds if, for every execution of* X, *every state in which* p *holds is followed by some later state in which* q *holds. In the third rule,* S *is any set of predicates.*

$$\frac{(p \wedge \neg q \textbf{ next } p \vee q).X \qquad \textbf{transient}.(p \wedge \neg q).X}{(p \leadsto q).X} \quad \text{(basis)}$$

$$\frac{(p \leadsto q).X \qquad (q \leadsto r).X}{(p \leadsto r).X} \quad \text{(transitivity)}$$

$$\frac{\langle \forall p \mid p \in S \rhd (p \leadsto q).X \rangle}{(\langle \exists p \mid p \in S \rhd p \rangle \leadsto q).X} \quad \text{(disjunction)}$$

This definition of the **leads-to** operator is due to Misra [46].

3.4.4 Follows

The **follows** operator, first described by Sivilotti [66], combines safety and progress properties and allows us to show the following relationship between two variables x and y of the same partially ordered type:

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

Typically, the partial ordering on x and y is clear from context. We may explicitly denote it by subscripting the **follows** operator with the ordering operator, as in $\textbf{follows}_{\leq}$ or $\textbf{follows}_{\preceq}$.

The **follows** operator for two variables x and y of the same partially ordered type and subsystem X is defined in terms of **stable**, $\leadsto$ and **next** as follows:

Definition 3.10 (Follows Operator)

$$\begin{aligned}(x \textbf{ follows } y).X \triangleq\ & \langle \forall k \rhd \textbf{stable}.(x \geq k).X \rangle \wedge \\ & \langle \forall k \rhd \textbf{stable}.(y \geq k).X \rangle \wedge \\ & \textbf{invariant}.(x \leq y).X \wedge \\ & \langle \forall k \rhd (y \geq k \leadsto x \geq k).X \rangle \wedge \\ & \langle \forall k \rhd (y = k \wedge x \leq k \textbf{ next } x \leq k).X \rangle\end{aligned}$$

3.5 The Channel Theorem

The channel theorem allows us to reason about message channels using the **follows** operator, by stating that a follows relationship holds for every inbox/process pair in a system.

Theorem 3.11 (Channel Theorem)
For all processes p, q *in a Dynamic UNITY system, and all inboxes* b *in* q*, the sequence of messages delivered to* q.b *from process* p *follows the sequence of messages sent by process* p *to* q.b*:*

$$\begin{aligned}\langle \forall p, q, b, i \mid p \in X.L_i \cup X.Z_i \wedge q \in X.L_i \cup X.Z_i \wedge b \in X.\mathcal{I}_i^q \rhd \\ ((X.\mathcal{I}^{q.b} \downarrow_m (m.\text{proc} = p)).\text{msg } \textbf{follows} \\ (X.\mathcal{O}^p \downarrow_m (m.\text{proc} = q \wedge m.\text{mbox} = b)).\text{msg}).X \rangle\end{aligned}$$

Proof

In order to prove the channel theorem, we must use the previously stated definition of the messaging system. The parts of the channel safety and progress properties from this definition that are relevant to the follows relationship in this theorem are the following:

$$\langle \forall i, b \mid 0 \leq i \wedge b \in \mathcal{I}_i \rhd \mathcal{I}_i^b \preceq \mathcal{I}_{i+1}^b \rangle \tag{a}$$

$$\langle \forall i \mid 0 \leq i \wedge l \in L_i \cup Z_i \rhd \mathcal{O}_i^l \preceq \mathcal{O}_{i+1}^l \rangle \tag{b}$$

$$\langle \forall p, q, i, b \mid p \in L_i \cup Z_i \wedge q \in L_i \cup Z_i \wedge 0 \leq i \wedge b \in \mathcal{I}_{i+1}^{q} \rhd (\mathcal{I}_{i+1}^{q.b} \downarrow_m (m.\text{proc} = p)).\text{msg} \preceq \mathcal{O}_i^p \downarrow_m (m.\text{proc} = q \wedge m.\text{mbox} = b).\text{msg} \rangle \quad \text{(c)}$$

$$\langle \forall p, q, i, b, k \mid p \in L_i \cup Z_i \wedge q \in L_i \cup Z_i \wedge 0 \leq i \wedge b \in \mathcal{I}_i^q \wedge k \preceq (\mathcal{O}_i^p \downarrow_m (m.\text{proc} = q \wedge m.\text{mbox} = b)).\text{msg} \rhd \langle \exists j \mid j > i \rhd k \preceq (\mathcal{I}_j^{q.b} \downarrow_m (m.\text{proc} = L)).\text{msg} \rangle \rangle \quad \text{(d)}$$

And the definition of x **follows** y is the following:

$$\begin{aligned}
(x \textbf{ follows } y).X \triangleq\ & \langle \forall k \rhd \textbf{stable}.(x \geq k).X \rangle \wedge & (1)\\
& \langle \forall k \rhd \textbf{stable}.(y \geq k).X \rangle \wedge & (2)\\
& \textbf{invariant}.(x \leq y).X \wedge & (3)\\
& \langle \forall k \rhd (y \geq k \rightsquigarrow x \geq k).X \rangle \wedge & (4)\\
& \langle \forall k \rhd (y = k \wedge x \leq k \textbf{ next } x \leq k).X \rangle & (5)
\end{aligned}$$

It is immediately clear that conjuncts (1) and (2) of the follows definition are satisfied by channel properties (a) and (b), respectively. Conjunct (3) of the follows definition is satisfied by channel properties (a) and (c). Property (c) says that the filtered $X.\mathcal{I}_{i+1}^{q.b}$ is always a subsequence of the filtered $X.\mathcal{O}_i^p$ and therefore, because of the monotonicity from property (a), so is the filtered $X.\mathcal{I}_i^{q.b}$. Conjunct (4) of the follows definition is satisfied by channel property (d), which says that every sequence exceeded by the filtered $X.\mathcal{O}^p$ is eventually exceeded by the filtered $X.\mathcal{I}^{q.b}$. Finally, it is immediately clear from the definition of **next** that conjunct (5) of the follows definition is satisfied by channel property (c). Therefore, the follows property holds. □

3.6 Other Useful Theorems

In this section we present some generally useful theorems about the operators we have defined, as well as some theorems about the functioning of Dynamic UNITY systems. Some of these theorems are immediately derivable from the definitions of the fundamental operators; we therefore do not present detailed proofs for all of them. In all of the following theorems, p, q, r, s, t are arbitrary predicates, and X is an arbitrary system.

3.6.1 Theorems about Next

The following theorems about **next** are due to Misra [47], and follow directly from the definitions of **next** and logical implication. Proofs for them will not be presented.

THEOREM 3.12

*Any predicate holds in all states immediately subsequent to states where the predicate **false** holds:*

$$(\textbf{false next } p).X$$

THEOREM 3.13

*The predicate **true** holds in all states immediately subsequent to states where any predicate holds:*

$$(p \textbf{ next true}).X$$

THEOREM 3.14 (CONJUNCTION AND DISJUNCTION)

The conjunction of any two **next** *properties gives additional next properties:*

$$(p \textbf{ next } q).X \wedge (r \textbf{ next } s).X \Rightarrow (p \wedge r \textbf{ next } q \wedge s).X \wedge (p \vee r \textbf{ next } q \vee s).X$$

THEOREM 3.15 (STRENGTHENING)

The left-hand side of any **next** *property can be strengthened:*

$$(p \textbf{ next } q).X \Rightarrow (p \wedge r \textbf{ next } q).X$$

THEOREM 3.16 (WEAKENING)

The right-hand side of any **next** *property can be weakened:*

$$(p \textbf{ next } q).X \Rightarrow (p \textbf{ next } q \vee r).X$$

3.6.2 Theorems about Transient

The following are theorems about the **transient** operator, which will be useful primarily for proving theorems about the $\leadsto$ operator later in this chapter. These theorems are due to Misra [46].

THEOREM 3.17 (STABILITY AND TRANSIENCE)

*The only predicate that is both stable and transient is **false**:*

$$(\textbf{stable}.p.X \wedge \textbf{transient}.p.X) \equiv [\neg p]$$

PROOF

The definition of **transient** tells us that **false** is transient, and Theorem 3.12 tells us that **false** is stable (because (**false next false**).X holds). Therefore, we only need to show that (**stable**.$p.X \wedge$ **transient**.$p.X) \Rightarrow [\neg p]$. We do this as follows:

From the definition of **transient**.$p.X$, there is some fair transition in X such that $\{p\}\ g \longrightarrow t\ \{\neg p\}$. From the definition of **stable**.$p.X$, $\{p\}\ g \longrightarrow t\ \{p\}$ for all fair transitions in X. Conjoining the postconditions of these two assertions gives $\{p\}\ g \longrightarrow t\ \{\textbf{false}\}$. Therefore, $\neg p$ must hold. □

THEOREM 3.18 (STRENGTHENING)

Any transient property can be strengthened:

$$\textbf{transient}.p.X \Rightarrow \textbf{transient}.(p \wedge q).X$$

PROOF

From the definition of **transient**.$p.X$, there is some fair transition in X such that $\{p\}\ g \longrightarrow t\ \{\neg p\}$. We can strengthen the right side and weaken the left side of this assertion, to give $\{p \wedge q\}\ g \longrightarrow t\ \{\neg p \vee \neg q\}$. This gives **transient**.$(p \wedge q).X$ by definition of **transient**. □

3.6.3 Theorems about Leads-to

The following are theorems about the $\leadsto$ operator. They are some of the most important theorems we will use to prove properties of Dynamic UNITY systems. These theorems are due to Misra [46].

THEOREM 3.19 (IMPLICATION)

Any implication is also a $\leadsto$ property:

$$[p \Rightarrow q] \Rightarrow (p \leadsto q).X$$

THEOREM 3.20 (STRENGTHENING)

The left-hand side of any $\leadsto$ property can be strengthened:

$$(p \leadsto q).X \Rightarrow (p \wedge r \leadsto q).X$$

THEOREM 3.21 (WEAKENING)

The right-hand side of any $\leadsto$ property can be weakened:

$$(p \leadsto q).X \Rightarrow (p \leadsto q \vee r).X$$

THEOREM 3.22 (DISJUNCTION)

A universally quantified set of $\leadsto$ properties can be transformed into a single $\leadsto$ property with existentially quantified operands. In this theorem, p_i *and* q_i *are predicates, and* i *is quantified over an arbitrary set:*

$$\langle \forall i \rhd (p_i \leadsto q_i).X \rangle \Rightarrow (\langle \exists i \rhd p_i \rangle \leadsto \langle \exists i \rhd q_i \rangle).X$$

THEOREM 3.23 (CANCELLATION)

Two $\leadsto$ properties of a particular form can be combined, cancelling an intermediate variable:

$$(p \leadsto q \vee r).X \wedge (r \leadsto s).X \Rightarrow (p \leadsto q \vee s).X$$

THEOREM 3.24 (IMPOSSIBILITY)

*A state satisfying **false** is reachable only from an unreachable state:*

$$(p \leadsto \textbf{\textit{false}}).X \Rightarrow [\neg p]$$

THEOREM 3.25 (PROGRESS-SAFETY-PROGRESS)

Progress and safety properties of specific forms can be combined to yield more complex progress properties:

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

3.6.4 Theorems about Follows

The following theorems about the **follows** operator are due to Sivilotti [66]. They are extremely important in proving relationships among communicating Dynamic UNITY processes. These theorems hold when the ordering relation used to define the **follows** operator defines a partially ordered set on the types of the variables.

THEOREM 3.26 (TRANSITIVITY)

*The **follows** operator is transitive:*

$$(x \textbf{ follows } y).X \wedge (y \textbf{ follows } z).X \Rightarrow (x \textbf{ follows } z).X$$

THEOREM 3.27 (REFLEXIVITY)

If a variable follows itself, its value never changes:

$$(x \textbf{ follows } x).X = \langle \exists k \rhd \textbf{invariant}.(x = k).X \rangle$$

THEOREM 3.28 (ANTISYMMETRY)

If two variables follow each other, their values are always equal:

$$(x \textbf{ follows } y).X \wedge (y \textbf{ follows } x).X \Rightarrow \textbf{invariant}.(x = y).X$$

THEOREM 3.29 (MONOTONICITY)

Application of a monotonic function to both sides of a ***follows*** *property preserves the* ***follows*** *property:*

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

THEOREM 3.30 (STABLE FIXED POINT)

An element k *is a fixed point of function* f *when* k = f.k. *We define the set* FP.f *of fixed points for a function* f *as follows:* FP.f = {k | k = f.k ▷ k}. *If a variable follows a monotonic function of itself, then it never changes once it reaches a fixed point of that function:*

$$(x \textbf{ follows } f.x).X \Rightarrow \langle \forall k \mid k \in FP.f \triangleright \textbf{stable}.(x = k).X \rangle$$

3.7 Verification of an Example Program

In this section, we present a small example program (Specification 3.2) and prove its correctness using the rules and theorems described earlier in this chapter. The program implements Euclid's algorithm for calculating the greatest common divisor (GCD) of two integers; it repeatedly reads an integer message from each of its two inboxes, performs Euclid's algorithm on these integers, and sends the result to a specified destination process and inbox.

Let GCDSeq be a function that takes two sequences of integers greater than or equal to 1 as input and produces as output a sequence of integers, with the same length as the shorter of the two input sequences, such that each element of the output sequence is the GCD of the corresponding elements of the two input sequences. The property we wish to prove about the GCD program is $\mathcal{O}$.msg **follows** GCDSeq(*xIn*.msg, *yIn*.msg) (we use *b*.msg to denote the sequence of messages on inbox/outbox *b*). The proof relies on the system constraint that all messages sent to *xIn* and *yIn* are positive integers. While we could have added conditions to the guards of GCDCalculator's transitions to filter out bad input messages, leaving them out reduces the complexity of the example and its proof.

In order to prove the follows property, we must first prove several other properties of GCDCalculator. We state these as lemmas, and prove some of them in a calculational fashion

program GCDCalculator(targetProcess: **process**, targetInbox: **string**)

declare
x, y: **integer**
xIn, yIn: **inbox**
waiting: **boolean**

always
busy $\triangleq \neg$waiting

initially
x = 1 $\wedge$ y = 1 $\wedge$ waiting

fair-transition
(1) xIn.**probe** $\wedge$ yIn.**probe** $\wedge$ waiting $\longrightarrow$
$x' =$ xIn.**current**.msg $\wedge$ $y' =$ yIn.**current**.msg $\wedge$
xIn.**advance** $\wedge$ yIn.**advance** $\wedge$ busy$'$
(2) [] x < y $\longrightarrow$ $y' = y - x$
(3) [] y < x $\longrightarrow$ $x' = x - y$
(4) [] x = y $\wedge$ busy $\longrightarrow$ **send**(targetProcess, targetInbox, x) $\wedge$ waiting$'$

end

Specification 3.2: The *GCDCalculator* program

by showing that they follow either directly from the predicate **true** or from our constraints on the system. For some lemmas we omit the calculational proofs, as the primary purpose of this example is to demonstrate the proof techniques for our various temporal operators and many of the lemmas use the same temporal operators. We present brief textual arguments for the correctness of lemmas for which we omit calculational proofs.

There is an implicit quantification over all possible instantiations of GCDCalculator for each property we state. That is, our properties hold for any instance of GCDCalculator regardless of its execution environment (with the system constraint that all messages sent to its inboxes are positive integers). We represent the GCD process within this implicit quantifier by *G*. In addition, we will often use a transition's number instead of the actual text of the transition when writing assertions within our proofs.

LEMMA 3.31 (POSITIVE INTEGERS)
The values of the state variables x *and* y *are always greater than or equal to 1.*

$$\textbf{invariant}.(x \geq 1 \wedge y \geq 1)$$

PROOF

$\mathbf{invariant}.(x \geq 1 \wedge y \geq 1)$

$=$ {definition of **invariant** operator}

$\mathbf{initially}.(x \geq 1 \wedge y \geq 1) \wedge (x \geq 1 \wedge y \geq 1)\ \mathbf{next}\ (x \geq 1 \wedge y \geq 1)$

$=$ {initially-section specifies $(x = 1 \wedge y = 1)$}

$(x \geq 1 \wedge y \geq 1)\ \mathbf{next}\ (x \geq 1 \wedge y \geq 1)$

$\Leftarrow$ {proof rule for **next** operator}

$\langle \forall s \mid s \in G.\mathcal{T}^+ (x \geq 1 \wedge y \geq 1) \rhd \{x \geq 1 \wedge y \geq 1\}\ s\ \{x \geq 1 \wedge y \geq 1\} \rangle$

$=$ {quantification}

$\{x \geq 1 \wedge y \geq 1\}\ (1)\ \{x \geq 1 \wedge y \geq 1\} \wedge \{x \geq 1 \wedge y \geq 1\}\ (2)\ \{x \geq 1 \wedge y \geq 1\} \wedge$
$\{x \geq 1 \wedge y \geq 1\}\ (3)\ \{x \geq 1 \wedge y \geq 1\} \wedge \{x \geq 1 \wedge y \geq 1\}\ (4)\ \{x \geq 1 \wedge y \geq 1\}$

We prove each conjunct individually:

$\{x \geq 1 \wedge y \geq 1\}\ (1)\ \{x \geq 1 \wedge y \geq 1\}$

$=$ {definition of assertion}

$(x \geq 1 \wedge y \geq 1 \wedge xIn.\mathbf{probe} \wedge yIn.\mathbf{probe} \wedge waiting \wedge x' = xIn.\mathbf{current} \wedge$
$\quad y' = yIn.\mathbf{current} \wedge xIn.\mathbf{advance} \wedge yIn.\mathbf{advance} \wedge busy' \Rightarrow x' \geq 1 \wedge y' \geq 1) \wedge$
$(x \geq 1 \wedge y \geq 1 \wedge \neg(xIn.\mathbf{probe} \wedge yIn.\mathbf{probe} \wedge waiting) \Rightarrow x \geq 1 \wedge y \geq 1)$

$=$ {predicate calculus: $x \wedge y \Rightarrow x$}

$x \geq 1 \wedge y \geq 1 \wedge xIn.\mathbf{probe} \wedge yIn.\mathbf{probe} \wedge waiting \wedge x' = xIn.\mathbf{current} \wedge$
$y' = yIn.\mathbf{current} \wedge xIn.\mathbf{advance} \wedge yIn.\mathbf{advance} \wedge busy' \Rightarrow x' \geq 1 \wedge y' \geq 1$

$\Leftarrow$ {predicate calculus: $(x \Rightarrow z) \Rightarrow (x \wedge y \Rightarrow z)$}

$x' = xIn.\mathbf{current} \wedge y' = yIn.\mathbf{current} \Rightarrow x' \geq 1 \wedge y' \geq 1$

$=$ {definition of **current** operation}

$x' = xIn[xIn.\mathbf{cnt}].\mathrm{msg} \wedge y' = yIn[yIn.\mathbf{cnt}].\mathrm{msg} \Rightarrow x' \geq 1 \wedge y' \geq 1$

$\Leftarrow$ {substitution}

$xIn[xIn.\mathbf{cnt}].\mathrm{msg} \geq 1 \wedge yIn[yIn.\mathbf{cnt}].\mathrm{msg} \geq 1$

$\Leftarrow$ {definition of message channels}

system constraint that all messages sent to *xIn* and *yIn* are positive integers

——————————————————————————— end of $\{x \geq 1 \wedge y \geq 1\}$ (1) $\{x \geq 1 \wedge y \geq 1\}$

$\{x \geq 1 \wedge y \geq 1\}$ (2) $\{x \geq 1 \wedge y \geq 1\}$

$=$ {definition of assertion}

$(x \geq 1 \wedge y \geq 1 \wedge x < y \wedge y' = y - x \Rightarrow x' \geq 1 \wedge y' \geq 1) \wedge$
$(x \geq 1 \wedge y \geq 1 \wedge y \leq x \Rightarrow x' \geq 1 \wedge y' \geq 1)$

$\Leftarrow$ {predicate calculus: $x \wedge y \Rightarrow x$}

$x \geq 1 \wedge y \geq 1 \wedge x < y \wedge y' = y - x \Rightarrow x' \geq 1 \wedge y' \geq 1$

$=$ {$x' = x$, since x' doesn't appear in transition (2)}

$x \geq 1 \wedge y \geq 1 \wedge x < y \wedge y' = y - x \Rightarrow x \geq 1 \wedge y' \geq 1$

$\Leftarrow$ {predicate calculus: $(x \Rightarrow y \wedge z) \Rightarrow (x \Rightarrow y)$}

$x \geq 1 \wedge y \geq 1 \wedge x < y \wedge y' = y - x \Rightarrow y' \geq 1$

$\Leftarrow$ {substitution}

$x \geq 1 \wedge y \geq 1 \wedge x < y \Rightarrow y - x \geq 1$

$=$ {arithmetic}

true

——— end of $\{x \geq 1 \wedge y \geq 1\}$ (2) $\{x \geq 1 \wedge y \geq 1\}$

$\{x \geq 1 \wedge y \geq 1\}$ (3) $\{x \geq 1 \wedge y \geq 1\}$

$\Leftarrow$ {symmetry with transition (2)}

true

——— end of $\{x \geq 1 \wedge y \geq 1\}$ (3) $\{x \geq 1 \wedge y \geq 1\}$

$\{x \geq 1 \wedge y \geq 1\}$ (4) $\{x \geq 1 \wedge y \geq 1\}$

$=$ {definition of assertion}

$(x \geq 1 \wedge y \geq 1 \wedge x = y \wedge \mathit{busy} \wedge \mathbf{send}(\mathit{targetProcess}, \mathit{targetInbox}, x) \wedge \mathit{waiting}' \Rightarrow$
$x' \geq 1 \wedge y' \geq 1) \wedge$
$(x \geq 1 \wedge y \geq 1 \wedge \neg(x = y \wedge \mathit{busy} \wedge \mathbf{send}(\mathit{targetProcess}, \mathit{targetInbox}, x)) \Rightarrow$
$x \geq 1 \wedge y \geq 1)$

$=$ {predicate calculus: $x \wedge y \Rightarrow x$}

$x \geq 1 \wedge y \geq 1 \wedge x = y \wedge \mathit{busy} \wedge \mathbf{send}(\mathit{targetProcess}, \mathit{targetInbox}, x) \wedge \mathit{waiting}' \Rightarrow$
$x' \geq 1 \wedge y' \geq 1$

$\Leftarrow$ {predicate calculus: $(x \Rightarrow z) \Rightarrow (x \wedge y \Rightarrow z)$}

$x \geq 1 \wedge y \geq 1 \Rightarrow x' \geq 1 \wedge y' \geq 1$

$\Leftarrow$ {$x' = x \wedge y' = y$, since x' and y' don't appear in transition (4)}

$x \geq 1 \wedge y \geq 1 \Rightarrow x \geq 1 \wedge y \geq 1$

= {predicate calculus: $x \Rightarrow x$}

true

——————————————————————————— end of $\{x \geq 1 \wedge y \geq 1\}$ (4) $\{x \geq 1 \wedge y \geq 1\}$

Therefore, the conjunction of all four assertions follows from the predicate **true** and our constraints on the system. □

LEMMA 3.32 (INBOX SYNCHRONIZATION)
The number of messages read from xIn *is always the same as the number of messages read from* yIn.

$$\textbf{invariant}.(xIn.\textbf{cnt} = yIn.\textbf{cnt})$$

PROOF

invariant.(xIn.**cnt** = yIn.**cnt**)

= {definition of **invariant** operator}

initially.(xIn.**cnt** = yIn.**cnt**) $\wedge$ (xIn.**cnt** = yIn.**cnt**) **next** (xIn.**cnt** = yIn.**cnt**)

= {all inboxes are initialized with **cnt** = 0}

(xIn.**cnt** = yIn.**cnt**) **next** (xIn.**cnt** = yIn.**cnt**)

⇐ {proof rule for **next** operator}

$\langle \forall s \mid s \in G.\mathcal{T}^+ (xIn.\textbf{cnt} = yIn.\textbf{cnt}) \rhd \{xIn.\textbf{cnt} = yIn.\textbf{cnt}\}\ s\ \{xIn.\textbf{cnt} = yIn.\textbf{cnt}\} \rangle$

= {quantification}

$\{xIn.\textbf{cnt} = yIn.\textbf{cnt}\}$ (1) $\{xIn.\textbf{cnt} = yIn.\textbf{cnt}\} \wedge \{xIn.\textbf{cnt} = yIn.\textbf{cnt}\}$ (2) $\{xIn.\textbf{cnt} = yIn.\textbf{cnt}\} \wedge$
$\{xIn.\textbf{cnt} = yIn.\textbf{cnt}\}$ (3) $\{xIn.\textbf{cnt} = yIn.\textbf{cnt}\} \wedge \{xIn.\textbf{cnt} = yIn.\textbf{cnt}\}$ (4) $\{xIn.\textbf{cnt} = yIn.\textbf{cnt}\}$

We prove each conjunct individually:

$\{xIn.\mathbf{cnt} = yIn.\mathbf{cnt}\}\ (1)\ \{xIn.\mathbf{cnt} = yIn.\mathbf{cnt}\}$

$=$ {definition of assertion}

$(xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \wedge xIn.\mathbf{probe} \wedge yIn.\mathbf{probe} \wedge waiting \wedge x' = xIn.\mathbf{current} \wedge$
$y' = yIn.\mathbf{current} \wedge xIn.\mathbf{advance} \wedge yIn.\mathbf{advance} \wedge busy' \Rightarrow xIn'.\mathbf{cnt} = yIn'.\mathbf{cnt}) \wedge$
$(xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \wedge \neg(xIn.\mathbf{probe} \wedge yIn.\mathbf{probe} \wedge waiting) \Rightarrow xIn.\mathbf{cnt} = yIn.\mathbf{cnt})$

$=$ {predicate calculus: $x \wedge y \Rightarrow x$}

$xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \wedge xIn.\mathbf{probe} \wedge yIn.\mathbf{probe} \wedge waiting \wedge x' = xIn.\mathbf{current} \wedge$
$y' = yIn.\mathbf{current} \wedge xIn.\mathbf{advance} \wedge yIn.\mathbf{advance} \wedge busy' \Rightarrow xIn'.\mathbf{cnt} = yIn'.\mathbf{cnt}$

$\Leftarrow$ {predicate calculus: $(x \Rightarrow z) \Rightarrow (x \wedge y \Rightarrow z)$}

$xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \wedge xIn.\mathbf{advance} \wedge yIn.\mathbf{advance} \Rightarrow xIn'.\mathbf{cnt} = yIn'.\mathbf{cnt}$

$=$ {definition of **advance** operation}

$xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \wedge xIn'.\mathbf{cnt} = xIn.\mathbf{cnt} + 1 \wedge yIn'.\mathbf{cnt} = yIn.\mathbf{cnt} + 1 \Rightarrow xIn'.\mathbf{cnt} = yIn'.\mathbf{cnt}$

$\Leftarrow$ {substitution}

$xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \Rightarrow xIn.\mathbf{cnt} + 1 = yIn.\mathbf{cnt} + 1$

$=$ {arithmetic}

true

---

end of $\{xIn.\mathbf{cnt} = yIn.\mathbf{cnt}\}\ (1)\ \{xIn.\mathbf{cnt} = yIn.\mathbf{cnt}\}$

$\{xIn.\mathbf{cnt} = yIn.\mathbf{cnt}\}\ (2)\ \{xIn.\mathbf{cnt} = yIn.\mathbf{cnt}\}$

$=$ {definition of assertion}

$(xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \wedge x < y \wedge y' = y - x \Rightarrow xIn'.\mathbf{cnt} = yIn'.\mathbf{cnt}) \wedge$
$(xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \wedge y \leq x \Rightarrow xIn.\mathbf{cnt} = yIn.\mathbf{cnt})$

$=$ {predicate calculus: $x \wedge y \Rightarrow x$}

$xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \wedge x < y \wedge y' = y - x \Rightarrow xIn'.\mathbf{cnt} = yIn'.\mathbf{cnt}$

$\Leftarrow$ {predicate calculus: $(x \Rightarrow z) \Rightarrow (x \wedge y \Rightarrow z)$}

$xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \Rightarrow xIn'.\mathbf{cnt} = yIn'.\mathbf{cnt}$

$=$ {$xIn'.\mathbf{cnt} = xIn.\mathbf{cnt} \wedge yIn'.\mathbf{cnt} = yIn.\mathbf{cnt}$, since xIn and yIn don't appear in transition (2)}

$xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \Rightarrow xIn.\mathbf{cnt} = yIn.\mathbf{cnt}$

$=$ {predicate calculus: $x \Rightarrow x$}

true

——— end of $\{xIn.\mathbf{cnt} = yIn.\mathbf{cnt}\}$ (2) $\{xIn.\mathbf{cnt} = yIn.\mathbf{cnt}\}$

$\{xIn.\mathbf{cnt} = yIn.\mathbf{cnt}\}$ (3) $\{xIn.\mathbf{cnt} = yIn.\mathbf{cnt}\}$

$\Leftarrow$ {symmetry with transition (2)}

true

——— end of $\{xIn.\mathbf{cnt} = yIn.\mathbf{cnt}\}$ (3) $\{xIn.\mathbf{cnt} = yIn.\mathbf{cnt}\}$

$\{xIn.\mathbf{cnt} = yIn.\mathbf{cnt}\}$ (4) $\{xIn.\mathbf{cnt} = yIn.\mathbf{cnt}\}$

$=$ {definition of assertion}

$(xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \wedge x = y \wedge busy \wedge \mathbf{send}(targetProcess, targetInbox, x) \wedge waiting' \wedge$
$\quad xIn'.\mathbf{cnt} = yIn'.\mathbf{cnt}) \wedge$
$(xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \wedge \neg(x = y \wedge busy) \Rightarrow xIn.\mathbf{cnt} = yIn.\mathbf{cnt})$

$=$ {predicate calculus: $x \wedge y \Rightarrow x$}

$xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \wedge x = y \wedge busy \wedge \mathbf{send}(targetProcess, targetInbox, x) \wedge waiting' \wedge$
$xIn'.\mathbf{cnt} = yIn'.\mathbf{cnt}$

$\Leftarrow$ {predicate calculus: $(x \Rightarrow z) \Rightarrow (x \wedge y \Rightarrow z)$}

$xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \Rightarrow xIn'.\mathbf{cnt} = yIn'.\mathbf{cnt}$

$=$ {$xIn'.\mathbf{cnt} = xIn.\mathbf{cnt} \wedge yIn'.\mathbf{cnt} = yIn.\mathbf{cnt}$, since xIn and yIn don't appear in transition (4)}

$xIn.\mathbf{cnt} = yIn.\mathbf{cnt} \Rightarrow xIn.\mathbf{cnt} = yIn.\mathbf{cnt}$

$=$ {predicate calculus: $x \Rightarrow x$}

true

——— end of {$xIn.\mathbf{cnt} = yIn.\mathbf{cnt}$} (4) {$xIn.\mathbf{cnt} = yIn.\mathbf{cnt}$}

Therefore, the conjunction of all four assertions follows from the predicate **true**. Having shown that they are equal, we refer to the values of $xIn.\mathbf{cnt}$ and $yIn.\mathbf{cnt}$ as *cnt* in the following lemmas. □

LEMMA 3.33 (VALUE TRANSIENCE)
If x *and* y *differ in a system state* S*, then in some system state subsequent to* S *either the value of* x*, the value of* y*, or both will have changed.*

$$\langle \forall X, Y \rhd \mathbf{transient}.(x \neq y \wedge x = X \wedge y = Y) \rangle$$

PROOF

$\langle \forall X, Y \rhd \mathbf{transient}.(x \neq y \wedge x = X \wedge y = Y) \rangle$

$\Leftarrow$ {proof rule for **transient** operator}

$\langle \forall X, Y \rhd \langle \exists s \mid s \in G.\mathcal{T}^{-}(x \neq y \wedge x = X \wedge y = Y) \rhd$
$\{x \neq y \wedge x = X \wedge y = Y\}\ s\ \{\neg(x \neq y \wedge x = X \wedge y = Y)\}\rangle\rangle$

$=$ {quantification}

$$\langle \forall X, Y \rhd \{x \neq y \wedge x = X \wedge y = Y\}\ (1)\ \{\neg(x \neq y \wedge x = X \wedge y = Y)\} \vee$$
$$\{x \neq y \wedge x = X \wedge y = Y\}\ (2)\ \{\neg(x \neq y \wedge x = X \wedge y = Y)\} \vee$$
$$\{x \neq y \wedge x = X \wedge y = Y\}\ (3)\ \{\neg(x \neq y \wedge x = X \wedge y = Y)\} \vee$$
$$\{x \neq y \wedge x = X \wedge y = Y\}\ (4)\ \{\neg(x \neq y \wedge x = X \wedge y = Y)\}\rangle$$

$\Leftarrow$ {predicate calculus: $x \Rightarrow x \vee y$}

$$\langle \forall X, Y \rhd \{x \neq y \wedge x = X \wedge y = Y\}\ (2)\ \{\neg(x \neq y \wedge x = X \wedge y = Y)\} \vee$$
$$\{x \neq y \wedge x = X \wedge y = Y\}\ (3)\ \{\neg(x \neq y \wedge x = X \wedge y = Y)\}\rangle$$

We simplify each disjunct individually:

$$\{x \neq y \wedge x = X \wedge y = Y\}\ (2)\ \{\neg(x \neq y \wedge x = X \wedge y = Y)\}$$

$=$ {definition of assertion}

$$(x \neq y \wedge x = X \wedge y = Y \wedge x < y \wedge y' = y - x \Rightarrow \neg(x' \neq y' \wedge x' = X \wedge y' = Y)) \wedge$$
$$(x \neq y \wedge x = X \wedge y = Y \wedge y \leq x \Rightarrow \neg(x \neq y \wedge x = X \wedge y = Y))$$

$=$ {predicate calculus: $x \Rightarrow y \equiv \neg x \vee y$}

$$(\neg(x \neq y \wedge x = X \wedge y = Y \wedge x < y \wedge y' = y - x) \vee \neg(x' \neq y' \wedge x' = X \wedge y' = Y)) \wedge$$
$$(\neg(x \neq y \wedge x = X \wedge y = Y \wedge y \leq x) \Rightarrow \neg(x \neq y \wedge x = X \wedge y = Y))$$

$=$ {De Morgan}

$$\neg((x \neq y \wedge x = X \wedge y = Y \wedge x < y \wedge y' = y - x \wedge x' \neq y' \wedge x' = X \wedge y' = Y) \vee$$
$$(x \neq y \wedge x = X \wedge y = Y \wedge y \leq x \wedge x \neq y \wedge x = X \wedge y = Y))$$

$\Leftarrow$ {Lemma 3.31, arithmetic: $x \geq 1 \wedge y \geq 1 \Rightarrow (y = Y \wedge y' = y - x \wedge y' = Y \equiv \textbf{false})$}

$$\neg(x \neq y \wedge x = X \wedge y = Y \wedge y \leq x \wedge x \neq y \wedge x = X \wedge y = Y)$$

$=$ {predicate calculus: $x \wedge x \equiv x$}

$$\neg(x \neq y \wedge x = X \wedge y = Y \wedge y \leq x)$$

$=$ {De Morgan}

$x = y \vee x \neq X \vee y \neq Y \vee x < y$

——————————————————— end of $\{x \neq y \wedge x = X \wedge y = Y\}$ (2) $\{\neg(x \neq y \wedge x = X \wedge y = Y)\}$

$\{x \neq y \wedge x = X \wedge y = Y\}$ (3) $\{\neg(x \neq y \wedge x = X \wedge y = Y)\}$

$\Leftarrow$ {symmetry with transition (2)}

$x = y \vee x \neq X \vee y \neq Y \vee y < x$

——————————————————— end of $\{x \neq y \wedge x = X \wedge y = Y\}$ (3) $\{\neg(x \neq y \wedge x = X \wedge y = Y)\}$

The disjunction of the two assertions is as follows:

$(x = y \vee x \neq X \vee y \neq Y \vee x < y) \vee (x = y \vee x \neq X \vee y \neq Y \vee y < x)$

$=$ {arithmetic: $x = y \vee x < y \vee y < x$}

true

Therefore, the disjunction of the two assertions follows from the predicate **true**. □

LEMMA 3.34 (CONSTRAINTS ON VALUE CHANGES)

If x *and* y *differ, then after the next system step either both* x *and* y *are unchanged or the sum of* x *and* y *is the maximum of the previous values of* x *and* y.

$$\langle \forall X, Y \rhd (x \neq y \wedge x = X \wedge y = Y) \textbf{ next } ((x \neq y \wedge x = X \wedge y = Y) \vee x + y = \max(X, Y)) \rangle$$

PROOF

This proof is similar in structure to the proofs of Lemmas 3.31 and 3.32 (after the decomposition into initially and next properties), so we will not present the calculations here. Note that the only two transitions that change x and y contain $x \neq y$ in their guard. If the guard of one of these transitions holds, the smaller of x and y is subtracted from the larger, and the sum of x and y in the subsequent state is the maximum of the previous values of x and y. Any other transition when $x \neq y$ preserves the values of x and y. Therefore, the next property holds. □

Lemma 3.35 (Calculation Progress, Part 1)
If x *and* y *differ in a system state* S*, then in some state subsequent to* S *the sum of* x *and* y *will be the maximum of the values of* x *and* y *in state* S*.*

$$\langle \forall X, Y \rhd (x \neq y \wedge x = X \wedge y = Y) \leadsto (x + y = \max(X, Y)) \rangle$$

Proof
Let $p \equiv x \neq y \wedge x = X \wedge y = Y$, $q \equiv x + y = \max(X, Y)$. Then we have:

$\langle \forall X, Y \rhd p \leadsto q \rangle$

$\Leftarrow$ {basis rule for $\leadsto$}

$\langle \forall X, Y \rhd (p \wedge \neg q \textbf{ next } p \vee q) \wedge \textbf{transient}.(p \wedge \neg q) \rangle$

$\Leftarrow$ {**next** and **transient** strengthening}

$\langle \forall X, Y \rhd (p \textbf{ next } p \vee q) \wedge \textbf{transient}.p \rangle$

$=$ {quantification}

$\langle \forall X, Y \rhd p \textbf{ next } p \vee q \rangle \wedge \langle \forall X, Y \rhd \textbf{transient}.p \rangle$

This predicate is exactly the conjunction of Lemmas 3.33 and 3.34. Therefore, the leads-to property follows from those lemmas. □

Lemma 3.36 (Calculation Progress, Part 2)
If x *and* y *differ in a system state* S*, then* x *and* y *will be equal in some state subsequent to* S*.*

$$\langle \forall K \mid x + y = K \rhd x \neq y \leadsto x = y \rangle$$

Proof
We prove this lemma inductively. For $K = 2$ (the base case), the only possible values for x and y are $x = y = 1$, since Lemma 3.31 specifies that x and y are positive integers. Thus, we have $1 \neq 1 \leadsto x = y$, or **false** $\leadsto x = y$, which holds by direct substitution into the basis rule for leads-to.

For subsequent values of K, assume the leads-to holds for all $2 \leq k < K$. There are two cases to consider: either $x = y = K/2$ or $x \neq y$. If $x = y = K/2$, the proof of the leads-to is

symmetric to that for the base case (**false** $\leadsto x = y$). If $x \neq y$, then Lemma 3.35 says that the sum of x and y will be reduced to $\max(x, y)$ in some later system state. Since $\max(x, y) < K$, the leads-to holds for $\max(x, y)$ by induction.

Therefore, the leads-to holds for all values of K greater than 1. □

LEMMA 3.37 (CALCULATION SAFETY)

Whenever at least one pair of integers has been read from the inboxes, the GCD of x *and* y *is equal to the GCD of the most recent pair of integers read from the inboxes.*

$$\textbf{invariant}.(cnt > 0 \Rightarrow \mathrm{GCD}(x, y) = \mathrm{GCD}(xIn[cnt - 1].\mathrm{msg}, yIn[cnt - 1].\mathrm{msg}))$$

PROOF

Observe that the only transition that modifies *cnt* is transition (1), which (using the value of *cnt* in the post-state) stores $xIn[cnt-1].$msg and $yIn[cnt-1].$msg in x and y, respectively. Observe also that the next property proven in Lemma 3.34 completely constrains the changes in x and y such that they implement Euclid's algorithm for GCD calculation. One of the characteristics of Euclid's algorithm is that, at every step, the GCD of the two intermediate values is the same as the GCD of the initial values. Therefore, the invariant holds. □

LEMMA 3.38 (MESSAGE COUNTS AND STATE, PART 1)

Assume that waiting *holds and the same number of messages have been read from each inbox as have been sent on the outbox. In the next state, either* waiting *still holds and the same number of messages have still been read from each inbox as have been sent on the outbox, or* busy *holds and one more message has been read from each inbox than has been sent on the outbox.*

$$\langle \forall C \mid C \geq 0 \rhd (waiting \wedge cnt = \mathcal{O}.\textbf{len} = C) \textbf{ next } ((waiting \wedge cnt = \mathcal{O}.\textbf{len} = C) \vee (busy \wedge cnt - 1 = \mathcal{O}.\textbf{len} = C)) \rangle$$

PROOF

Assume $waiting \wedge cnt = \mathcal{O}.\textbf{len} = C$ holds. The only enabled transition that can change either *waiting* or *cnt* is transition (1), and no transition that can change *out*.**len** is enabled. If any transition other than (1) is selected, $waiting \wedge cnt = \mathcal{O}.\textbf{len} = C$ is maintained. If transition (1) is selected, it establishes *busy* and increments *cnt* by 1 as a result of reading messages from the inboxes. Therefore, after execution of transition (1), $busy \wedge cnt - 1 = \mathcal{O}.\textbf{len} = C$ holds. This proves the next property. □

LEMMA 3.39 (MESSAGE COUNTS AND STATE, PART 2)

Assume that busy *holds and one more message has been read from each inbox than has been sent on the outbox. In the next state, either* busy *still holds and one more message has still been*

read from each inbox than has been sent on the outbox, or waiting *holds and the same number of messages have been read from each inbox as have been sent on the outbox.*

$$\langle \forall C \mid C \geq 0 \rhd (\mathit{busy} \wedge \mathit{cnt} - 1 = \mathcal{O}.\mathbf{len} = C)\ \mathbf{next}$$
$$((\mathit{busy} \wedge \mathit{cnt} - 1 = \mathcal{O}.\mathbf{len} = C) \vee (\mathit{waiting} \wedge \mathit{cnt} = \mathcal{O}.\mathbf{len} = C + 1))\rangle$$

PROOF

Assume $\mathit{busy} \wedge \mathit{cnt} - 1 = \mathcal{O}.\mathbf{len} = C$ holds. The only enabled transition that can change either *busy* or *out*.**len** is transition (4), and no transition that can change *cnt* is enabled. If any transition other than (4) is selected, $\mathit{busy} \wedge \mathit{cnt} - 1 = \mathcal{O}.\mathbf{len} = C$ is maintained. If transition (4) is selected, it establishes *waiting* and increments *out*.**len** by 1 as a result of sending a message on the outbox. Therefore, after execution of transition (4), $\mathit{waiting} \wedge \mathit{cnt} = \mathcal{O}.\mathbf{len} = C + 1$ holds. This proves the next property. □

LEMMA 3.40 (CALCULATION TERMINATION)

If busy holds in a system state S *and* x *and* y *differ in state* S*, then* x *and* y *will be equal to each other and to the GCD of the most recent pair of integers read from the inboxes in some state subsequent to* S*.*

$$\langle \forall C \mid C \geq 0 \rhd (\mathit{busy} \wedge \mathit{cnt} - 1 = C \wedge x \neq y) \rightsquigarrow$$
$$(\mathit{busy} \wedge \mathit{cnt} - 1 = C \wedge x = y = \mathrm{GCD}(\mathit{xIn}[C].\mathrm{msg}, \mathit{yIn}[C].\mathrm{msg}))\rangle$$

PROOF

This property follows from an application of the basis rule for leads-to with Lemmas 3.36 and 3.39. The invariant proven in Lemma 3.37 shows that x and y always have the same GCD as $\mathit{xIn}[C].\mathrm{msg}$ and $\mathit{yIn}[C].\mathrm{msg}$, so when $x = y$, their value must by definition be that GCD. □

LEMMA 3.41 (COMMUNICATION SAFETY, PART 1)

If waiting *holds in a system state where there are messages waiting on both* xIn *and* yIn*, then in the next system state either this remains the case or* busy *holds and a message has been read from each inbox.*

$$\langle \forall C \mid C \geq 0 \rhd (\mathit{waiting} \wedge \mathit{cnt} = C \wedge \mathit{xIn}.\mathbf{probe} \wedge \mathit{yIn}.\mathbf{probe})\ \mathbf{next}$$
$$((\mathit{waiting} \wedge \mathit{cnt} = C \wedge \mathit{xIn}.\mathbf{probe} \wedge \mathit{yIn}.\mathbf{probe}) \vee$$
$$(\mathit{busy} \wedge \mathit{cnt} - 1 = C \wedge x = \mathit{xIn}[C].\mathrm{msg} \wedge y = \mathit{yIn}[C].\mathrm{msg}))\rangle$$

PROOF

Assume *waiting* holds and there are messages waiting on both *xIn* and *yIn*. The only enabled transition is transition (1), and its effect is to establish *busy* and (by means of the **current** and **advance** operations) set x and y to *xIn*[C].msg and *yIn*[C].msg, respectively, and increment *cnt*. This proves the next property. □

LEMMA 3.42 (WAITING TRANSIENCE)

If waiting *holds in a system state* S *where there are messages waiting on both* xIn *and* yIn*, then* waiting *will be falsified in some system state subsequent to* S.

$$\textbf{transient}.(\mathit{waiting} \wedge \mathit{xIn}.\textbf{probe} \wedge \mathit{yIn}.\textbf{probe})$$

PROOF

Assume *waiting* holds and messages are waiting on *xIn* and *yIn*. The only enabled transition is transition (1), which falsifies *waiting*. It must eventually be selected because of the weak fairness requirement on program execution. Therefore, the transient property holds. □

LEMMA 3.43 (COMMUNICATION PROGRESS, PART 1)

If waiting *holds in a system state* S *where there are messages waiting on* xIn *and* yIn*, then in some system state subsequent to* S *the next messages on* xIn *and* yIn *will have been read.*

$$\langle \forall C \mid C \geq 0 \rhd (\mathit{waiting} \wedge cnt - 1 = C \wedge \mathit{xIn}.\textbf{probe} \wedge \mathit{yIn}.\textbf{probe}) \leadsto (\mathit{busy} \wedge cnt - 1 = C \wedge \mathrm{GCD}(x, y) = \mathrm{GCD}(\mathit{xIn}[C].\mathrm{msg}, \mathit{yIn}[C].\mathrm{msg})) \rangle$$

PROOF

This property follows from an application of the basis rule for leadsto with Lemmas 3.41 and 3.42. □

LEMMA 3.44 (COMMUNICATION SAFETY, PART 2)

If busy *holds in a system state where* x *and* y *are equal to each other and to the GCD of the most recent pair of integers read from the inboxes, then in the next state either this remains the case or* waiting *holds and the GCD has been sent on the outbox.*

$$\langle \forall C \mid C \geq 0 \rhd (\mathit{busy} \wedge x = y = \mathrm{GCD}(\mathit{xIn}[C].\mathrm{msg}, \mathit{yIn}[C].\mathrm{msg})) \ \textbf{next} \ ((\mathit{busy} \wedge x = y = \mathrm{GCD}(\mathit{xIn}[C].\mathrm{msg}, \mathit{yIn}[C].\mathrm{msg})) \vee (\mathit{waiting} \wedge x = y = \mathcal{O}[C].\mathrm{msg} = \mathrm{GCD}(\mathit{xIn}[C].\mathrm{msg}, \mathit{yIn}[C].\mathrm{msg})) \rangle$$

PROOF

Assume *busy* holds and $x = y$. The only enabled transition is transition (4), and its effect is to establish *waiting* and (by means of the **send** operation) set $\mathcal{O}[C]$.msg to x. This proves the next property. □

LEMMA 3.45 (BUSY TRANSIENCE)

If busy *holds in a system state* S *where* x *and* y *are equal to each other, then* busy *will be falsified in some system state subsequent to* S.

$$\textbf{transient}.(busy \wedge x = y)$$

PROOF

Assume *busy* holds and $x = y$. The only enabled transition is transition (4), which falsifies *busy*. It must eventually be selected because of the weak fairness requirement on program execution. Therefore, the transient property holds. □

LEMMA 3.46 (COMMUNICATION PROGRESS, PART 2)

If busy *holds in a system state* S *where* x *and* y *are equal to each other and to the GCD of the most recent pair of integers read from the inboxes, then in some system state subsequent to* S *the GCD will have been sent on the outbox.*

$$\langle \forall C \mid C \geq 0 \rhd (busy \wedge x = y = \mathrm{GCD}(xIn[C].\mathrm{msg}, yIn[C].\mathrm{msg})) \leadsto (waiting \wedge x = y = \mathcal{O}[C].\mathrm{msg} = \mathrm{GCD}(xIn[C].\mathrm{msg}, yIn[C].\mathrm{msg}))\rangle$$

PROOF

This property follows from an application of the basis rule for leads-to with Lemmas 3.44 and 3.45. □

LEMMA 3.47 (INITIAL CONDITIONS)

In the initial state, waiting *holds and the same number of messages have been read from each inbox as have been sent on the outbox.*

$$\textbf{initially}.(waiting \wedge cnt = \mathcal{O}.\textbf{len} = 0)$$

PROOF

The initially property follows immediately from the initially-section of GCDCalculator and the fact that all mailboxes are empty at initialization. □

THEOREM 3.48 (GCD CALCULATION AND COMMUNICATION)
The sequence of integers on the outbox follows the sequence containing the GCDs of pairs of messages on the inboxes.

$$\mathcal{O}.\text{msg} \textbf{ follows } \text{GCDSeq}(xIn.\text{msg}, yIn.\text{msg})$$

PROOF
In order to prove this follows relation, we must prove the five components of the follows definition. For this follows property, they are:

$$\langle \forall s \rhd \textbf{stable}.(\mathcal{O}.\text{msg} \geq s) \rangle \tag{1}$$

$$\langle \forall s \rhd \textbf{stable}.(\text{GCDSeq}(xIn.\text{msg}, yIn.\text{msg}) \geq s) \rangle \tag{2}$$

$$\textbf{invariant}.(\mathcal{O}.\text{msg} \leq \text{GCDSeq}(xIn.\text{msg}, yIn.\text{msg})) \tag{3}$$

$$\langle \forall s \rhd \text{GCDSeq}(xIn.\text{msg}, yIn.\text{msg}) \geq s \leadsto \mathcal{O}.\text{msg} \geq s \rangle \tag{4}$$

$$\langle \forall s \rhd \text{GCDSeq}(xIn.\text{msg}, yIn.\text{msg}) = s \wedge \mathcal{O}.\text{msg} \leq s \textbf{ next } \mathcal{O}.\text{msg} \leq s \rangle \tag{5}$$

Since *xIn* and *yIn* are inboxes, *xIn*.msg and *yIn*.msg are monotonic by definition. The application of GCDSeq to them is therefore monotonic. This proves component (1) of the definition.

Since $\mathcal{O}$ is an outbox, $\mathcal{O}$.msg is monotonic by definition. This proves component (2) of the definition.

Initially, both the outbox and the inboxes are empty, so $\mathcal{O}$.msg is initially a subsequence of GCDSeq(*xIn*.msg, *yIn*.msg). By Lemmas 3.38-3.39, 3.43, and 3.46, the GCD of every pair of messages read from the inboxes will eventually be written to the outbox in the order the messages are read, and the number of messages written to the outbox is never more than one fewer than the number of messages read from each inbox. Therefore, every write to the outbox preserves the invariant in component (3) of the definition, as $\mathcal{O}$.msg remains a subsequence of GCDSeq(*xIn*.msg, *yIn*.msg). The fact that every pair of messages will eventually be processed proves component (4) of the definition, and the fact that the number of messages written to the outbox never exceeds the number of messages read from each inbox proves component (5) of the definition.

Therefore, the follows property holds, and we have proven our desired property. □

In the following three chapters, we will present systems composed of multiple Dynamic UNITY programs. We will omit most of the calculations in our proofs of these systems, as those proofs are similar in structure to the ones we have presented here.

Chapter 4

Deterministic Example: The Prime Number Sieve

We now present the first of our example Dynamic UNITY systems, Eratosthenes' prime number sieve. This example is chosen for its determinism; we know the computations that must be done to generate the set of prime numbers, so we can easily check that our system implements the algorithm correctly. Our particular system illustrates one of Dynamic UNITY's strengths, by generating an unbounded number of running, communicating processes while still ensuring progress.

4.1 Problem Statement

The problem statement for this example is straightforward: "Generate an infinite sequence of integers such that the ith element of the sequence is the ith prime number." We know of an algorithm to generate arbitrarily large prime numbers: Eratosthenes' prime number sieve. We can use Dynamic UNITY's communication and process creation capabilities to create an infinitely large prime number sieve, and thereby generate our infinite sequence (provided that we allow our system to run for infinite time).

We choose to implement each sieve element as its own Dynamic UNITY process, with knowledge only about the previous and next elements in the sieve. We wish to implement a Dynamic UNITY system that actually generates the infinite sequence of prime numbers, so we also need a process that feeds integers into the sieve. We choose to have a central "collection point" for the prime numbers, so that we can keep our generated sequence in one place. This allows us to reason about the sequence and its changes, rather than reasoning simultaneously about the states of an infinite number of processes. For simplicity, and to preserve the ordering of the sequence, the process that acts as the collection point and the process that feeds integers into the sieve are the same process.

For the purposes of this discussion, we assume that the only messages being sent in our system are integer messages. This minimizes our proof obligations and makes the programs less complicated, because we don't need to worry about filtering out non-integer messages that may arrive on the inboxes.

We specify the sieve element program first, describing and proving properties of its behavior in isolation, and then do the same for the sequence generator and collector program. Finally, we compose the two programs to create a prime number sieve.

4.2 The Sieve Program

---

program Sieve(previous: **process**, sieveValue: **integer**, sequenceNumber: **integer**)

declare
numberIn, primeIn: **inbox**
next: **process**

always
remainder $\triangleq$ numberIn.**current**.msg **mod** sieveValue

initially
send(previous, "primeIn", sieveValue) $\wedge$ next = $\perp$

fair-transition
(1) primeIn.**probe** $\longrightarrow$
send(previous, "primeIn", primeIn.**current**.msg) $\wedge$ primeIn.**advance**
(2) [] numberIn.**probe** $\wedge$ remainder = 0 $\longrightarrow$ numberIn.**advance**
(3) [] numberIn.**probe** $\wedge$ remainder $\neq$ 0 $\wedge$ next = $\perp$ $\longrightarrow$
next$'$ = **new** Sieve(**this**, numberIn.**current**.msg, sequenceNumber + 1)
(4) [] numberIn.**probe** $\wedge$ remainder $\neq$ 0 $\wedge$ next $\neq$ $\perp$ $\longrightarrow$
send(next, "numberIn", numberIn.**current**.msg) $\wedge$ numberIn.**advance**

end

Specification 4.1: The *Sieve* program, part of the infinite prime number sieve system

---

The Sieve program acts as a filter, taking two integer sequences as inputs (from the *numberIn* and *primeIn* inboxes) and generating outputs based on these sequences and its instantiation parameters. Additionally, each Sieve program is responsible for creating the next Sieve program in the system. In order to show that the Sieve program filters these sequences correctly, we must establish certain properties; we first prove that the Sieve program is well-formed:

THEOREM 4.1 (SIEVE WELL-FORMEDNESS)
The Sieve program is well-formed.

PROOF

We prove that the Sieve program is well-formed by showing that its always-section, initially-section, and transition-sections are well-formed:

always-section The always-section of the Sieve program contains a single definition, which is a simple function of 2 integers. There are no circular definitions, and no undefined variables are referenced. Therefore, the always-section is well-formed.

initially-section The initially-section of the Sieve program sends a message and initializes *next* to the empty process. Neither of these actions is capable of causing an infinite recursion, and both of them are satisfiable. Therefore, the initially-section is well-formed.

transition-sections For each transition, we must demonstrate that its postcondition is always satisfiable if its precondition holds. We do this individually for the four transitions:

- The postcondition of transition (1) is the conjunction of a message send (which is always satisfiable) and an inbox advance (which is always satisfiable). Therefore, transition (1)'s postcondition is always satisfiable.
- The postcondition of transition (2) is an inbox advance, which is always satisfiable.
- The postcondition of transition (3) is a process instantiation, which is always satisfiable.
- The postcondition of transition (4) is the conjunction of a message send (which is always satisfiable) and an inbox advance (which is always satisfiable). Therefore, transition (4)'s postcondition is always satisfiable.

Since all sections of the Sieve program are well-formed, the entire program is well-formed.

□

We now prove safety and progress properties that show that the Sieve process filters the integer sequences as necessary to implement Eratosthenes' algorithm. We introduce the function SIEVE for use in the following proofs. This function takes as parameters a sequence of integers and a single integer, and generates as a result the sequence of integers that was passed to it with all elements divisible by the single integer parameter removed. We also use the notation "*box*.msg" to refer to the sequence of messages contained in inbox or outbox *box* (that is, the sequence "*box*[0].msg, *box*[1].msg, …, *box*[*box*.**len** - 1].msg").

THEOREM 4.2 (SIEVE NUMBER OUTPUT SEQUENCE)

The sequence of messages sent to inboxes named "numberIn" follows the sequence of messages received on numberIn*, with integers divisible by* sieveValue *removed from the sequence.*

$$(\mathcal{O} \downarrow_m m.\text{mbox} = \text{“numberIn”}).\text{msg } \textbf{follows } \text{SIEVE}(\textit{numberIn}.\text{msg}, \textit{sieveValue})$$

PROOF

To prove this follows relation, we must prove the five components of the follows definition. In the following equations and their proofs, we denote SIEVE(*numberIn*.msg, *sieveValue*) by *numberInSieved* and $(\mathcal{O} \downarrow_m m.\text{mbox} = \text{“numberIn”})$ by *numberOut*:

$$\langle \forall s \rhd \textbf{stable}.(\textit{numberOut}.\text{msg} \geq s) \rangle \tag{1}$$

$$\langle \forall s \rhd \textbf{stable}.(\textit{numberInSieved} \geq s) \rangle \tag{2}$$

$$\textbf{invariant}.(\textit{numberOut}.\text{msg} \leq \textit{numberInSieved}) \tag{3}$$

$$\langle \forall s \rhd (\textit{numberInSieved} \geq s) \rightsquigarrow (\textit{numberOut}.\text{msg} \geq s) \rangle \tag{4}$$

$$\langle \forall s \rhd (\textit{numberInSieved} = s \wedge \textit{numberOut}.\text{msg} \leq s) \textbf{ next } (\textit{numberOut}.\text{msg} \leq s) \rangle \tag{5}$$

We address these individually:

1. Since *numberOut* is a filtered outbox, *numberOut*.msg is monotonic by definition.

2. Since *numberIn* is an inbox, *numberIn*.msg is monotonic by definition. Additionally, *sieveValue* does not change during execution because it is a parameter to the Sieve program. This implies that *numberInSieved* is monotonic, because SIEVE always removes the same elements of a monotonic sequence.

3. Two transitions, (2) and (4), perform advance operations on *numberIn*; the other transitions do not modify the state of either *numberIn* or $\mathcal{O}$, and can therefore not affect this invariant). Transition (2) advances *numberIn* to the next message and does nothing with the current message; it executes only when *remainder* is 0, which occurs when the current message is evenly divisible by *sieveValue*. Transition (4) advances *numberIn* to the next message and appends the current message to *numberOut* (by sending a message to an inbox named "numberIn"); it executes only when *remainder* is not 0, which occurs when the current message is not evenly divisible by *sieveValue*. Since both of these commands are guarded by *numberIn*.**probe**, and both contain advance operations, exactly one of them executes for each message in inbox *numberIn*. The SIEVE function removes exactly those elements of the sequence that are evenly divisible by its parameter, in this case *sieve-*

Value, which is exactly what transition (2) does. Therefore, *numberOut*.msg is always a subsequence of *numberInSieved*.

4. From the previous equation, we know that *numberOut*.msg is always a subsequence of *numberInSieved*. Weak fairness ensures that all 4 transitions of the Sieve program—and in particular, transition (4)—will be selected infinitely often. As long as *numberOut*.msg $\neq$ *numberInSieved*, transition (4) will at some point execute when selected (transition (2) may execute a finite number of times first), extending *numberOut*.msg. Therefore, if at some point *numberInSieved* is a supersequence of a sequence s, *numberOut*.msg will become a supersequence of s after some finite number of executions of (4).

5. The only way for the sequence *numberOut*.msg to increase is by means of transition (4), which reads a message from *numberIn* and sends it to an inbox named "numberIn." Therefore, *numberOut*.msg can never exceed the previous state of *numberInSieved*.

All five components of the original follows relation hold. Therefore, the follows relation holds. □

THEOREM 4.3 (SIEVE PRIME OUTPUT SEQUENCE)
The sequence of messages sent to inboxes named "primeIn" follows the sequence of messages received on primeIn*, with* sieveValue *prepended to the sequence.*

$$(\mathcal{O} \downarrow_m m.\text{mbox} = \text{“primeIn”}).\text{msg } \textbf{follows } (\mathit{sieveValue} \bowtie \mathit{primeIn}.\text{msg})$$

PROOF
To prove this follows relation, we must prove the following 5 components of the follows definition. In the following equations and their proofs, we denote (*sieveValue* $\bowtie$ *primeIn*.msg) by *extendedPrimeIn* and ($\mathcal{O} \downarrow_m m$.mbox = "primeIn") by *primeOut*, for brevity:

$$\langle \forall s \rhd \textbf{stable}.(\mathit{primeOut}.\text{msg} \geq s) \rangle \tag{1}$$

$$\langle \forall s \rhd \textbf{stable}.(\mathit{extendedPrimeIn} \geq s) \rangle \tag{2}$$

$$\textbf{invariant}.(\mathit{primeOut}.\text{msg} \leq \mathit{extendedPrimeIn}) \tag{3}$$

$$\langle \forall s \rhd (\mathit{extendedPrimeIn} \geq s) \leadsto (\mathit{primeOut}.\text{msg} \geq s) \rangle \tag{4}$$

$$\langle \forall s \rhd (\mathit{extendedPrimeIn} = s \wedge \mathit{primeOut}.\text{msg} \leq s) \textbf{ next } (\mathit{primeOut}.\text{msg} \leq s) \rangle \tag{5}$$

We address these individually:

1. Since *primeOut* is a filtered outbox, *primeOut*.msg is monotonic by definition.

2. Since *primeIn* is an inbox, *primeIn*.msg is monotonic by definition. Additionally, *sieveValue* does not change during execution, because it is a parameter to the Sieve program. Therefore, *extendedPrimeIn* is monotonic.

3. Only transition (1) and the initialization of the Sieve program change *primeOut*.msg. At initialization, *sieveValue* is sent to an inbox named "primeIn" (and thus added to *primeOut*). Since *primeOut*.msg is monotonic, this means that the first element of *primeOut*.msg is always *sieveValue*. Transition (1) takes the next message from *primeIn* and appends it to *primeOut* (by sending it to an inbox named "primeIn"), and advances *primeIn*; it executes exactly once for each message in *primeIn*, and appends each message from *primeIn* to *primeOut* in the order it was received. Therefore, *primeOut*.msg is always a subsequence of *extendedPrimeIn*.

4. From the previous equation, we know that *primeOut*.msg is always a subsequence of *extendedPrimeIn*. Weak fairness ensures that all 4 transitions of the Sieve program—and in particular, transition (1)—will be selected infinitely often. As long as *primeOut*.msg $\neq$ *extendedPrimeIn*, transition (1) will execute when selected, extending *primeOut*.msg. Therefore, if at some point *extendedPrimeIn* is a supersequence of a sequence s, *primeOut*.msg will become a supersequence of s after some finite number of executions of (1).

5. The only way for the sequence *primeOut*.msg to increase is by means of transition (1), which reads a message from *primeIn* and sends it to an inbox named "primeIn." Therefore, *primeOut*.msg can never exceed the previous state of *extendedPrimeIn*.

All five components of the follows definition hold. Therefore, the follows relation holds. □

Finally, we prove properties that show that the Sieve program creates the next Sieve process, with appropriate parameters, when necessary.

THEOREM 4.4 (NEXT SIEVE PROCESS STABILITY)
Once next *refers to a process, it refers to the same process forever.*

$$\langle \forall p \mid p \neq \perp \rhd \textbf{stable}.(next = p) \rangle$$

PROOF
Initially, $next = \perp$. There is only one transition, (3), which assigns a value to *next* by creating a new process. Transition (3) is guarded by $next = \perp$; it can therefore only execute once during the process' entire execution, because thereafter *next* will have a non-$\perp$ value. Since there are no other transitions that change the value of *next*, the stable properties hold. □

THEOREM 4.5 (NEXT SIEVE PROCESS EXISTENCE)

If one or more messages have been sent to an inbox called "numberIn," there exists a next sieve process.

$$\textbf{invariant}.(|\mathcal{O} \downarrow_m m.\text{mbox} = \text{“numberIn”}| > 0 \Rightarrow \mathit{next} \neq \bot)$$

PROOF

There is only one transition, (4), which sends a message to an inbox called "numberIn." This transition is guarded by $\mathit{next} \neq \bot$, so no message is sent to an inbox called "numberIn" unless this guard holds.

From Theorem 4.4, we know that $\mathit{next} \neq \bot$ is stable. Therefore, if any message has been sent to an inbox called "numberIn," the condition $\mathit{next} \neq \bot$ holds. This proves our implication. □

THEOREM 4.6 (NEXT SIEVE PROCESS CREATION SAFETY)

At all times during the execution of the Sieve program, exactly one of the following three conditions holds: (a) No next process has been created, and no number that is not evenly divisible by sieveValue *has been received on* numberIn*; (b) No next process has been created, and at least one number that is not evenly divisible by* sieveValue *has been received on* numberIn*; or (c) A next process has been created with its* sieveValue *parameter set to the first number received on* numberIn *that is not evenly divisible by* sieveValue.

invariant.

$$\begin{array}{l}
((\mathit{next} = \bot \wedge \\
\quad \langle \forall n \mid 0 \leq n < \mathit{numberIn}.\textbf{len} \rhd \mathit{numberIn}[n].\text{msg } \textbf{mod } \mathit{sieveValue} = 0\rangle) \vee \qquad \text{(a)} \\
(\mathit{next} = \bot \wedge \\
\quad \langle \exists n \mid 0 \leq n < \mathit{numberIn}.\textbf{len} \rhd \mathit{numberIn}[n].\text{msg } \textbf{mod } \mathit{sieveValue} \neq 0\rangle \wedge \qquad \text{(b)} \\
\quad \mathit{numberIn}.\textbf{cnt} \leq \langle \min\ n \mid \mathit{numberIn}[n].\text{msg } \textbf{mod } \mathit{sieveValue} = 0 \rhd n\rangle) \vee \\
(\mathit{next} = \\
\quad \text{Sieve}(\mathit{primeIn}, \\
\qquad \mathit{numberIn}[\langle \min\ n \mid \mathit{numberIn}[n].\text{msg } \textbf{mod } \mathit{sieveValue} = 0 \rhd n\rangle], \qquad \text{(c)} \\
\qquad \mathit{sequenceNumber} + 1) \wedge \\
\quad \mathit{numberIn}.\textbf{cnt} > \langle \min\ n \mid \mathit{numberIn}[n].\text{msg } \textbf{mod } \mathit{sieveValue} = 0 \rhd n\rangle))
\end{array}$$

PROOF

The three disjuncts of the invariant are mutually exclusive, by inspection. Initially, disjunct (a)

holds, because the *numberIn* inbox is empty and the variable *next* is initialized to $\perp$.

We now show that each of the possible transitions maintains the invariant. Since the invariant contains no reference to *primeIn*, transition (1) always maintains the invariant, because it only modifies *primeIn*. We therefore ignore transition (1) for the remainder of this proof.

When disjunct (a) holds, the only transition that can be executed is (2), since there is no message in *numberIn* that yields a remainder when divided by *sieveValue*. This transition maintains disjunct (a), because it does not change the contents of *numberIn* (it merely changes *numberIn*.**cnt**). Additionally, when disjunct (a) holds, a message can be received on *numberIn*. If this message is evenly divisible by *sieveValue*, disjunct (a) is maintained; otherwise, disjunct (b) is established because there now exists a message in *numberIn* that is not evenly divisible by *sieveValue*.

When disjunct (b) holds, the only transitions that can be executed are (2) and (3), which are mutually exclusive. If (2) is executed, disjunct (b) is maintained, because *next* is unchanged. If (3) is executed, disjunct (c) is established because execution of (3) reads the first value in *numberIn* that is not evenly divisible by *sieveValue* and assigns to *next* a new Sieve process whose parameters are exactly those specified in disjunct (c).

When disjunct (c) holds, the only transitions that can be executed are (2) and (4). Both of these maintain (c), because neither changes *next* and because of the monotonicity of inbox and outbox message sequences.

We have now shown that the invariant holds, because it holds initially and every transition takes the system from a state where one of the disjuncts holds to another state where one of the disjuncts holds. □

THEOREM 4.7 (NEXT SIEVE PROCESS CREATION PROGRESS)

If a number that is not evenly divisible by sieveValue *is received on* numberIn*, the next Sieve process will be created in finite time.*

$$\textbf{transient}.(\mathit{next} = \perp \wedge \langle \exists n \mid 0 \leq n < \mathit{numberIn}.\textbf{len} \rhd \mathit{numberIn}[n].\text{msg } \textbf{mod } \mathit{sieveValue} \neq 0 \rangle)$$

PROOF

The state we wish to show transience of corresponds to disjunct (b) of Theorem 4.6. Recall that the only transitions that can be executed in this state are (2) and (3), which are mutually exclusive. Since some element of *numberIn* exists that is not divisible by *sieveValue*, let N be the index of that element, and let $\mathit{numberIn}.\textbf{cnt} = n \leq N$. If transition (2) is executed, *numberIn*.**cnt** is incremented from its pre-execution value, which must be strictly less than N. N remains unchanged, because of the monotonicity of inbox message sequences. Therefore, weak fairness ensures that *numberIn*.**cnt** will, after a finite number of executions of transition

(2), be incremented to N. When *numberIn*.**cnt** = N, the precondition of transition (3) holds, and it must therefore be executed in accordance with weak fairness. Execution of transition (3) invalidates disjunct (b) of Theorem 4.6, which proves the transient property. □

We have now shown all the behavior of the Sieve in isolation that we need to prove the correctness of the composed system.

4.3 The Generator Program

---

program Generator

declare
primeIn: **inbox**
sequenceNumber: **integer**
next: **process**
num: **integer**
primes: **sequence** {**integer**}

initially
primes = Λ ∧ num = 2 ∧ next = **new** Sieve(**this**, 2, 1) ∧ sequenceNumber = 0

fair-transition
(1) **send**(next, "numberIn", num) ∧ num′ = num + 1
(2) [] primeIn.**probe** ⟶ primes′ = primes ⋈ primeIn.**current**.msg ∧ primeIn.**advance**

end

Specification 4.2: The *Generator* program, part of the infinite prime number sieve system

The functions of the Generator program are to feed the sequence of positive integers starting from 2 into the prime number sieve, and to accept the sequence of prime numbers from the sieve. Additionally, the Generator program is responsible for creating the first Sieve process (corresponding to 2, the first prime number). For the purpose of proving properties of the Generator in isolation, we assume that a well-formed program named Sieve which takes 3 initialization parameters (a process and two integers) and which creates no processes during its initialization exists in the same system as the Generator, without making any other assumptions about that program's specification. We first prove that the Generator program is well-formed.

THEOREM 4.8 (GENERATOR WELL-FORMEDNESS)
The Generator program is well-formed.

PROOF

We prove that the Generator program is well-formed by showing that its always-section, initially-section, and transition-sections are well-formed:

always-section The Generator program has no always-section.

initially-section The initially-section of the Generator program instantiates a Sieve process. To show that the Generator program's initially-section is well-formed, we must show that this instantiation is not infinitely recursive and that the initially-section as a whole is satisfiable. The Sieve program's initially-section does not instantiate any processes, so there is no recursion. Moreover, all conjuncts of the Generator program's initially-section are satisfiable. Therefore, the Generator program's initially-section is well-formed.

transition-sections For each transition, we must demonstrate that its postcondition is always satisfiable if its precondition holds. We do this individually for the two transitions:

- The postcondition of transition (1) is the conjunction of a variable increment and a message send, both of which are always satisfiable. Therefore, transition (1)'s postcondition is always satisfiable.
- The postcondition of transition (2) is the conjunction of an append of a received message to a sequence and an inbox advance operation, both of which are always satisfiable. Therefore, transition (2)'s postcondition is always satisfiable.

Since all sections of the Generator program are well-formed, the entire program is well-formed. □

We now prove safety and progress properties that show that the sequence sent by the Generator program on its outbox is exactly the sequence of positive integers starting from 2, and that the number of integers sent always increases.

THEOREM 4.9 (GENERATOR OUTPUT SEQUENCE SAFETY)

The outbox of a Generator process contains a sequence of integer messages beginning with the number 2:

$$\textbf{invariant}.(\langle \forall\, i \mid 0 \le i < \mathcal{O}.\textbf{len} \rhd \mathcal{O}[i].\text{msg} = 2 + i \rangle)$$

In order to prove this theorem, we need to establish a relation between $\mathcal{O}.\textbf{len}$ and *num*. We do so with the following lemma:

LEMMA 4.10 (RELATION BETWEEN $\mathcal{O}$.LEN AND *num*)

For all Generator processes, the difference between the value of integer variable num *and the number of messages in the process's outbox is always 2:*

$$\textbf{invariant}.(num - \mathcal{O}.\textbf{len} = 2)$$

PROOF

Initially, $num = 2$ and $\mathcal{O}.\textbf{len} = 0$ (because an outbox contains no messages at initialization, and there are no message sends in the initially-section of program Generator). Therefore, $num - \mathcal{O}.\textbf{len} = 2$ holds immediately after process initialization.

Transition (1) contains a send operation that sends a single message, incrementing $\mathcal{O}.\textbf{len}$ by 1 due to the semantics of the message passing system, and explicitly increments *num* by 1. Since incrementing two integers by the same value doesn't change their difference, $num - \mathcal{O}.\textbf{len} = 2$ holds immediately after an execution of transition (1) if it held before the execution.

Transition (2) contains no message sends and no references to *num*, and therefore does not affect the invariant. □

PROOF OF THEOREM 4.10 (GENERATOR OUTPUT SEQUENCE SAFETY)

Initially, $\mathcal{O}.\textbf{len} = 0$, so the quantification is over an empty range ($0 \leq i < 0$) and holds vacuously. The only transition that changes $\mathcal{O}.\textbf{len}$ is transition (1), which (owing to the semantics of **send**) assigns to *numberOut[numberOut.**len**].msg* the value of the integer variable *num*, and increments both $\mathcal{O}.\textbf{len}$ and *num*. From Lemma 4.10, we know that $num - \mathcal{O}.\textbf{len} = 2$ before every execution of transition (1). We can therefore conclude that the value of each message in *numberOut* is its index in *numberOut* plus 2. □

THEOREM 4.11 (GENERATOR OUTPUT SEQUENCE PROGRESS)

The outbox of a Generator process always has a new message added to it in finite time:

$$\langle \forall i \mid i \geq 0 \rhd \textbf{transient}.(\mathcal{O}.\textbf{len} = i) \rangle$$

PROOF

The Generator program only has two transitions. The weak fairness property ensures that transition (1) will always be executed after transition (2) has been executed a finite number of times and, as we established in the proof of Lemma 4.10, transition (1) increments the value of $\mathcal{O}.\textbf{len}$ by 1. Since the Generator program does not contain a **stop** statement, it runs forever. This means that no matter what the value of $\mathcal{O}.\textbf{len}$ is at a given point in the program's execution, it will always be incremented at some point later in the execution. □

Next, we show that the sequence *primes* follows the sequence of messages received on channel *primeIn*.

THEOREM 4.12 (GENERATOR INPUT SEQUENCE)
The sequence primes *follows the message history of* primeIn.

$$\textit{primes}\ \textbf{follows}\ \textit{primeIn}.\text{msg}$$

PROOF
To prove this follows relation, we must prove the following 5 components of the follows definition:

$$\langle \forall s \rhd \textbf{stable}.(\textit{primes} \geq s) \rangle \tag{1}$$

$$\langle \forall s \rhd \textbf{stable}.(\textit{primeIn}.\text{msg} \geq s) \rangle \tag{2}$$

$$\textbf{invariant}.(\textit{primes} \leq \textit{primeIn}.\text{msg}) \tag{3}$$

$$\langle \forall s \rhd (\textit{primeIn}.\text{msg} \geq s) \leadsto (\textit{primes} \geq s) \rangle \tag{4}$$

$$\langle \forall s \rhd (\textit{primeIn}.\text{msg} = s \wedge \textit{primes} \leq s)\ \textbf{next}\ (\textit{primes} \leq s) \rangle \tag{5}$$

We address these individually:

1. There is one transition which appends elements to *primes*, and there are no transitions that remove elements from *primes*. Therefore, *primes* is monotonic.

2. Since *primeIn* is an inbox, *primeIn*.msg is monotonic by definition.

3. One transition, (2), performs an advance operation on *primeIn*. When it does this, it also appends the current message to *primes*. Since this transition is guarded by *primeIn*.**probe**, it executes exactly once for each message on *primeIn*. Both *primeIn*.msg and *primes* are initially empty sequences (and therefore subsequences of each other), and this transition always maintains that subsequence relationship.

4. From the previous equation, we know that *primes* is always a subsequence of *primeIn*.msg. Weak fairness ensures that both transitions of the Sieve program—and in particular, transition (2)—will be selected infinitely often. As long as $\textit{primes} \neq \textit{primeIn}.\text{msg}$, transition (2) will at some point execute when selected, extending *primes*. Therefore, if at some point *primeIn*.msg is a supersequence of a sequence s, *primes* will become a supersequence of s after some finite number of executions of (2).

5. The only way for the sequence *primes* to increase is by means of transition (2), which reads a message from *primeIn* and appends it to *primes*. Therefore, *primes* can never exceed the previous state of *primeIn*.msg.

All five components of the follows definition hold. Therefore, the follows relation holds. □

Finally, we prove that the process referred to by *next* remains the same throughout the execution of the Generator program.

THEOREM 4.13 (GENERATOR NEXT REFERENCE STABILITY)
Once a process is instantiated and a reference to that process is assigned to next *in the Generator program,* next *refers to that process forever:*

$$\langle \forall p \triangleright \textbf{stable}.(next = p) \rangle$$

PROOF
Initially, *next* is assigned to be a reference to a new instance of the Sieve program with a specific set of parameters. No transition in the Generator program modifies the *next* reference. Therefore, *next* refers to the same instance of the Sieve program forever. □

Finally, we show that the Generator's *sequenceNumber* variable is always set to 0. Though this does not seem like an interesting property in isolation, it will prove useful when we construct the proof of the composed system.

THEOREM 4.14 (GENERATOR SEQUENCE NUMBER)
The sequenceNumber *variable of a Generator process is always set to 0.*

$$\textbf{invariant}.(sequenceNumber = 0)$$

PROOF
At initialization, *sequenceNumber* is set to 0. No transition of the Generator program changes *sequenceNumber*. Therefore, *sequenceNumber* is always set to 0.

We have now shown all the behavior of the Generator in isolation that we will need to use in our proof of correctness for the composed system.

4.4 The Composed System

The composed system contains the Generator program (Specification 4.2) as an initial program, as well as the Sieve program (Specification 4.1). We now prove that the entire system actually implements a prime number sieve, by proving three theorems about the system using the properties we have already proven about the Generator and Sieve programs in isolation.

system InfinitePrimeNumberSieve

initial-program Generator

program Sieve(previous: **process**, sieveValue: **integer**, sequenceNumber: **integer**)

end

Specification 4.3: The *InfinitePrimeNumberSieve* system

THEOREM 4.15 (GENERATOR PROCESS UNIQUENESS/STABILITY)
There is always exactly one Generator process in the InfinitePrimeNumberSieve system, and this process never stops.

PROOF
We need to prove that a single Generator process is created in every possible execution of the system, that it never stops, and that it is impossible for any more to be created after it. Generator is the initial program of the system, so a single instance of Generator is created at system initialization time. Theorem 4.8 shows that the Generator program is well-formed, and by inspection it does not contain a **stop** statement. Therefore, in accordance with Dynamic UNITY execution semantics, it runs forever. Also by inspection, neither the Generator program nor the Sieve program contains a statement that instantiates a Generator process. This means that no other Generator process is ever instantiated during the execution of the system. □

THEOREM 4.16 (GLOBAL SAFETY)
At all times, the primes *sequence contained in the Generator process is comprised of the first* primes.**length** *prime numbers in increasing order.*

In order to prove this theorem, we must show first that the *sieveValue* parameters of the Sieve processes in the system are prime numbers, and then that the *primes* sequence consists of these *sieveValue* parameters, in order. We first prove several lemmas that allow us to construct a proof of global safety.

LEMMA 4.17 (SIEVE UNIQUENESS)
For any sequence number n*, there is at most one Sieve process in the system with that sequence number.*

invariant.$(\langle \forall p, q, u, v, w, x, y, z \mid$
$p, q \in \mathcal{P} \wedge p = \text{Sieve}(u, v, w) \wedge q = \text{Sieve}(x, y, z) \rhd (w = z) \Rightarrow (p = q) \rangle)$

PROOF

Sieve processes are created in one of two ways: by the Generator at initialization time, or by a Sieve process after initialization time. Theorem 4.4 tells us that when a Sieve process creates another Sieve process, that process is stable (that is, no other Sieve process is ever created by the creating Sieve process), while Theorem 4.13 tells us the same for the Generator process. Therefore, all we need to show is that every time a Sieve process is created, it is created using a sequence number that has not been used before. This follows directly from the program text of the Sieve program, since transition (3) of Sieve increments the sequence number when creating a new Sieve process. □

LEMMA 4.18 (SIEVE SEQUENCE)

The sequence numbers of the Sieve processes in the system accurately reflect their position in the chain of Sieve processes.

$$\begin{aligned}\textbf{invariant}.(\langle \forall s, p, v, n \mid s \in \mathcal{P} \wedge s = \text{Sieve}(p, v, n) \rhd \\ p.sequenceNumber = n - 1 \wedge \\ (s.next = \bot \vee (s.next \neq \bot \wedge s.next.sequenceNumber = n + 1))\rangle)\end{aligned}$$

PROOF

The Generator, which is the first process to be created in the system, always has *sequenceNumber* = 0, by Theorem 4.14. As part of the Generator's initialization, it creates a Sieve process with *sequenceNumber* = 1, with itself as the Sieve's previous process. Therefore, after creation of the first Sieve process s, the invariant holds: $s.sequenceNumber = 1$, $s.previous.sequenceNumber = 0$, and $s.next = \bot$.

We now show, by induction, that all subsequent Sieve process creations maintain the invariant. Assume we have n Sieve processes numbered k, $k+1$, ..., $k+n-1$ in the system, and denote the nth Sieve process by $S(n)$ (so we have the relation $S(n).sequenceNumber = k+n-1$). The invariant, if it holds, must hold with $S(n).next = \bot$ (if it doesn't, there is an $(n+1)$st Sieve process we are not accounting for). If transition (3) of $S(n)$ is not executed, the invariant is maintained because no new Sieve process is created and $S(n).next$ is not changed. If transition (3) is executed, a new Sieve process $S(n+1)$ is created such that $S(n+1).sequenceNumber = S(n).sequenceNumber + 1$ and $S(n+1).next = \bot$. This preserves the invariant, since $S(n).next = S(n+1)$ and $S(n+1).previous = S(n)$, and we have now proven the invariant for $n+1$ Sieve processes. As the validity of our base case, $n = 1$, has been shown above, this completes the induction. □

Corollary 4.19 (Sieve Connections)

Every Sieve process's previous *process has that Sieve as a* next *process.*

$$\textbf{invariant}.(\langle \forall s, p, v, n \mid s \in \mathcal{P} \wedge s = \text{Sieve}(p, v, n) \triangleright p.next = s \rangle)$$

Proof

Combining Lemmas 4.17 and 4.18 immediately gives us this corollary - together, they show that there is a single chain of Sieve processes in the system, in which each Sieve is connected exactly to its predecessor and, if one exists, its successor. □

Lemma 4.20 (Messaging Connections)

For every Sieve process S*, the message sequence of inbox* S.numberIn *follows the sequence of messages sent on* $\mathcal{O}^{\text{S.previous}}$ *to inboxes named "numberIn," and the message sequence of inbox* S.previous.primeIn *follows the sequence of messages sent on* $\mathcal{O}^{\text{S}}$ *to inboxes named "primeIn."*

$$\begin{aligned}\langle \forall s, p, v, n \mid s \in \mathcal{P} \wedge s = \text{Sieve}(p, v, n) \triangleright \\ s.numberIn.\text{msg } \textbf{follows } (\mathcal{O}^p \downarrow_m m.\text{mbox} = \text{“numberIn”}).\text{msg} \wedge \\ p.primeIn.\text{msg } \textbf{follows } (\mathcal{O}^s \downarrow_m m.\text{mbox} = \text{“primeIn”}).\text{msg}\rangle\end{aligned}$$

Proof

By inspection, all messages sent by any process, Generator or Sieve, to an inbox named "numberIn" are sent to the inbox with that name belonging to the *next* process. Because of Lemmas 4.17 and 4.18, we can also tell by inspection that no messages are sent to that inbox from any other outbox belonging to any process in the system, and that the "numberIn" inbox of a Sieve process is always the variable *numberIn*. Therefore, by the channel theorem (Theorem 3.11), the sequence of messages in the *numberIn* inbox of a Sieve process follows the sequence of messages in the outbox of its predecessor which are addressed to inboxes named "numberIn." An analogous proof applies to the sequence of messages in the outbox of a Sieve process which are addressed to inboxes named "primeIn" and its predecessor's *primeIn* inbox. □

Corollary 4.21 (Sieve Process Outgoing Number Sequences)

For every Sieve process S*, the sequence of messages sent on* $\mathcal{O}^{\text{S}}$ *to inboxes named "numberIn" follows the sequence of messages sent on* $\mathcal{O}^{\text{S.previous}}$*, with all integers divisible by* S.sieveValue *removed.*

$$\langle \forall s, p, v, n \mid s \in \mathcal{P} \wedge s = \text{Sieve}(p, v, n) \rhd$$
$$(\mathcal{O}^s \downarrow_m m.\text{mbox} = \textit{“numberIn”}).\text{msg } \textbf{follows}$$
$$\text{SIEVE}((\mathcal{O}^p \downarrow_m m.\text{mbox} = \textit{“numberIn”}).\text{msg}, v)\rangle$$

PROOF

This corollary follows immediately from Lemma 4.20 and Theorem 4.2. □

Hereafter, we will denote $(\mathcal{O}^k \downarrow_m m.\text{mbox} = \text{“numberIn”})$ by $k.\textit{numberOut}$, and $(\mathcal{O}^k \downarrow_m m.\text{mbox} = \text{“primeIn”})$ by $k.\textit{primeOut}$, for brevity.

LEMMA 4.22 (PRIME SIEVE VALUES)

The sieveValue *of the Sieve process with sequence number* n *is the* n*th prime number.*

PROOF

We prove this by induction on the sequence numbers of the Sieve processes, using our knowledge of the initial Sieve's parameters, the output sequence of the Generator, and the rules governing prime numbers. For sequence number 1, the lemma holds because the Sieve initialized with sequence number 1 is initialized with sieve value 2, and 2 is the first prime number. This is our base case.

Now, assume we have n Sieve processes $S_1, S_2, \ldots, S_n$, with sequence numbers 1, 2, ..., n and sieve values p_1, p_2, ..., p_n, such that for all i, p_i is the ith prime number. We need to show that when Sieve process S_{n+1} is created (with sequence number $n + 1$), its sieve value p_{n+1} is the $(n + 1)$st prime number. From Corollary 4.21, we know that $S_n.\textit{numberOut}.\text{msg}$ **follows** $\text{SIEVE}(S_{n-1}.\textit{numberOut}.\text{msg}, p_n)$. This relation also applies to the *numberOut*.msg sequence of every other Sieve process. Therefore, all elements of $S_{n-1}.\textit{numberOut}.\text{msg}$ are not divisible by any of $p_1, p_2, \ldots, p_{n-1}$. Additionally, we know from Theorem 4.9 that the *numberOut*.msg sequence of the Generator, which is the *previous* process to S_1, is exactly the sequence 2, 3, ... Therefore, the *numberOut*.msg sequence of S_n is exactly the sequence which results when all elements divisible by p_1 are removed from the initial sequence, then all elements divisible by p_2 are removed from the resulting sequence, etcetera, all the way up to all elements divisible by p_n being removed from $S_n.\textit{numberIn}.\text{msg}$. This is exactly the Eratosthenes' prime number sieve algorithm.

Since p_n is prime, the first element of $S_n.\textit{numberOut}.\text{msg}$—that is, the first element of $S_{n-1}.\textit{numberOut}.\text{msg}$ not divisible by p_n—must be prime as well, because a number that is not divisible by any prime number less than itself is prime. Moreover, it is the first prime number greater than p_n, because of the particular integer sequence being fed into the system by the

Generator. This element becomes p_{n+1}, the *sieveValue* for process S_{n+1}, as shown in Theorem 4.6. Since $p_1, p_2, \ldots, p_n$ are the 1st, 2nd, ..., nth prime numbers, p_{n+1} is the $(n+1)$st prime number. □

COROLLARY 4.23 (SIEVE PROCESS PRIME SEQUENCES)

For every Sieve process S*, the message sequence of* S.primeOut *follows the message sequence of* S.next.primeOut *with* S.sieveValue *prepended.*

$$\langle \forall s, p, v, n \mid s \in \mathcal{P} \wedge s = \text{Sieve}(p, v, n) \rhd$$
$$s.primeOut.\text{msg } \textbf{follows } (v \bowtie s.next.primeOut.\text{msg}) \rangle$$

PROOF

This corollary follows immediately from Lemma 4.20 and Theorem 4.3.

PROOF OF THEOREM 4.16 (GLOBAL SAFETY)

The *primes* sequence in the Generator process follows *primeOut* of the Sieve process with *sequenceNumber* 1, by Lemma 4.20 and Theorems 4.12 and 4.14. This sequence, by Lemma 4.22 and repeated application of Corollary 4.23, is exactly the sequence of prime numbers starting from 2. □

THEOREM 4.24 (GLOBAL PROGRESS)

The primes *sequence never remains the same length forever.*

$$\langle \forall n \mid n \geq 0 \rhd (\text{primes}.\textbf{length} = n) \leadsto (\text{primes}.\textbf{length} > n) \rangle$$

PROOF

Global progress follows immediately from Theorems 4.11, 4.12 and 4.3 and Corollaries 4.21 and 4.23. □

Chapter 5

Nondeterministic Example: Single Resource Mutual Exclusion

Our second example Dynamic UNITY system is a simple mutual exclusion algorithm, where clients request a single resource from a central server and are served in the order of their requests. We demonstrate the ability of Dynamic UNITY processes to leave the system, by allowing clients to leave the system at any time during their execution. We also make use of unfair transitions, another feature of Dynamic UNITY, in this example.

5.1 Problem Statement

The problem statement for this example is as follows: "Given a shared resource and a changing set of clients that require access to the resource, ensure that only one client at a time has access to the resource and that no client that requests access to the resource is forced to wait forever for that access." The most straightforward way of accomplishing this is to have a request queue, so that requests for the resource are processed in the order in which they are received. Dynamic UNITY gives us such a queuing mechanism, the inbox, as part of its messaging framework.

We implement the resource (more accurately, the process that manages access to the resource) as a single Dynamic UNITY process, and each client as a Dynamic UNITY process. Access to the resource is controlled by a *token*, which is reflected in the states of the resource and the clients: the resource is holding a token when its *idle* definition holds, and the client is holding a token when its *busy* definition holds. Every message passed in the system contains either one token or no tokens; there are no multiple-token messages.

In addition to the basic mutual exclusion algorithm, we give the clients the ability to leave the system at any time during their execution to better simulate a real resource allocation system (in which a consumer may decide to abandon a request for a resource if that request is not serviced within a reasonable time).

We specify the components of our system individually, describe and prove properties of their behavior in isolation, and then compose them to solve the problem.

5.2 The Resource Program

---

program Resource

declare
requestIn, releaseIn: **inbox**
releases: **multiset** {**process**}
current: **process**

always
idle $\triangleq$ current = $\perp$;
busy $\triangleq$ $\neg$idle

initially
current = $\perp$ $\wedge$ releases = $\varnothing$

fair-transition
(1) idle $\wedge$ requestIn.**probe** $\longrightarrow$
requestIn.**advance** $\wedge$ current$'$ = requestIn.**current**.proc $\wedge$
send(requestIn.**current**.proc, "tokenIn", $\varnothing$)
(2) [] busy $\wedge$ current $\in$ releases $\longrightarrow$ current$'$ = $\perp$ $\wedge$ releases$'$ = releases \ {current}
(3) [] releaseIn.**probe** $\longrightarrow$
releaseIn.**advance** $\wedge$ releases$'$ = releases $\cup$ {releaseIn.**current**.proc}

end

Specification 5.1: The *Resource* program, part of the single resource mutual exclusion system

The Resource program is responsible for ensuring mutually exclusive access to a resource, by handling requests and releases sent to appropriately-named inboxes and sending tokens to appropriate destinations. In isolation, we can prove that the Resource program is well-formed, that it always gets a token back before sending another token, and that its message histories and other state variables fulfill certain other restrictions that will be important to the proof of the composed system.

THEOREM 5.1 (RESOURCE WELL-FORMEDNESS)
The Resource program is well-formed.

PROOF
We prove that the Resource program is well-formed by showing that its always-section, initially-section, and transition-sections are well-formed:

always-section The always-section of the Resource program contains two definitions. The first is a simple Boolean function of a variable and a constant, and the second is the negation of the first. These definitions are not circular, and no undefined variables are referenced. Therefore, the always-section is well-formed.

initially-section The initially-section of the Resource program initializes a set to the empty set, and a process to $\perp$. There is no recursion, and both conjuncts of the initially predicate are satisfiable. Therefore, the initially-section is well formed.

transition-sections For each transition, we must demonstrate that its postcondition is always satisfiable if its precondition holds. We do this individually for the three transitions:

- The postcondition of transition (1) is the conjunction of an inbox advance, an assignment to a variable and a message send. All of these are always satisfiable. Therefore, transition (1)'s postcondition as a whole is always satisfiable.
- The postcondition of transition (2) is the conjunction of an assignment to a variable and a set difference operation, both of which are always satisfiable. Therefore, transition (2)'s postcondition is always satisfiable.
- The postcondition of transition (3) is the conjunction of an inbox advance and a set union operation, both of which are always satisfiable. Therefore, transition (3)'s postcondition is always satisfiable.

Since all sections of the Resource program are well-formed, the entire program is well-formed. □

THEOREM 5.2 (RESOURCE PROCESS SERVING SAFETY)
A Resource only serves one process at a time. That is, if current *refers to a process, then* current *always changes to* $\perp$ *before changing to refer to a different process.*

$$\langle \forall c \mid c \neq \perp \rhd current = c \ \ \textbf{next} \ \ (current = c \vee current = \perp) \rangle$$

PROOF
When $current = c \neq \perp$, the only enabled transitions are (2) and (3). Transition (2) changes *current* to $\perp$, so it satisfies the next properties. Transition (3) does not change *current*, so it also satisfies the next properties. Additionally, changes in inbox states due to arriving messages do not change *current*. Therefore, the next properties hold. □

THEOREM 5.3 (RESOURCE PER-PROCESS MESSAGE HISTORIES)

For every process in the system that is not the process referred to by current*, the difference between the number of releases a Resource has received from that process and the number of tokens it has sent to that process is exactly the number of times that process appears in the* releases *multiset. For the process referred to by* current *when a Resource is busy, the difference between the number of releases the Resource has received from that process and the number of tokens it has sent to that process is exactly the number of times that process appears in the* releases *multiset minus 1.*

$$\textbf{invariant}.(\langle \forall\, p \mid p \in \mathcal{P} \wedge p \neq \mathit{current} \rhd (\mathit{releaseIn} \downarrow_m m.\mathrm{proc} = p).\textbf{cnt} - (\mathcal{O} \downarrow_m m.\mathrm{proc} = p).\textbf{len} = \langle \#q \mid q \in \mathit{releases} \rhd q = p \rangle \rangle) \quad \text{(a)}$$

$$\textbf{invariant}.(\mathit{busy} \Rightarrow (\mathit{releaseIn} \downarrow_m m.\mathrm{proc} = \mathit{current}).\textbf{cnt} - (\mathcal{O} \downarrow_m m.\mathrm{proc} = \mathit{current}).\textbf{len} = \langle \#p \mid p \in \mathit{releases} \rhd p = \mathit{current} \rangle - 1 \rangle) \quad \text{(b)}$$

PROOF

Initially, both invariants hold because *current* is initialized to $\perp$ and all mailbox histories are empty, giving $0 - 0 = 0$, a tautology, for all instances of (a) and **false** $\Rightarrow 0 - 0 = -1$, which simplifies to **true**, for (b). We now show that each transition maintains both invariants.

Transition (1) is enabled only when $\mathit{current} = \perp$, so for all processes p in the system ($\perp$ can never be an actual process), an instance of invariant (a) holds as a precondition. Transition (1) sends a single message to a particular process and sets *current* to refer to that process (falsifying *idle* and establishing *busy*); it does not change the *releases* multiset or the *releaseIn* mailbox state. By inspection, it maintains (b) by establishing both its left and right sides while maintaining (a) for all processes other than the new *current*.

Transition (2) is enabled only when *busy* holds, so it executes with the right side of invariant (b) as a precondition. It decreases the number of instances of *current* in the *releases* multiset by 1, and establishes *idle* (falsifying *busy*). Invariant (a) is maintained, because the former *current* now satisfies (a) (by substitution), and invariant (b) is maintained because *busy* is falsified.

Transition (3) advances *releaseIn*, receiving a message from a process p, and also increases the number of instances of p in the *releases* multiset by 1. It therefore maintains both invariants, by simple arithmetic. □

THEOREM 5.4 (RESOURCE PER-PROCESS REQUEST/TOKEN SAFETY)
For every process in the system, the number of token messages sent by a Resource to that process is exactly the number of requests the Resource has read from that process.

$$\textbf{invariant}.(\langle \forall\, p \mid p \in \mathcal{P} \rhd (\mathcal{O} \downarrow_m m.\text{proc} = p).\textbf{len} = (\mathit{requestIn} \downarrow_m m.\text{proc} = p).\textbf{cnt} \rangle)$$

PROOF
Initially, the invariant holds because the Resource's mailboxes are all empty. There is only one transition, (1), which sends messages, and it also is the only transition that receives messages on *requestIn*. Every time it executes, it receives a single message on *requestIn* and sends a single message to the process that sent the message it received on *requestIn*. Therefore, both sides of the invariant's equality are incremented by 1 every time transition (1) executes. Since no other transition changes either side of the equality, the invariant is maintained. □

THEOREM 5.5 (RESOURCE REQUEST HANDLING PROGRESS)
If a Resource is idle and there is a request waiting in its requestIn *inbox, the process that sent the request will be served.*

$$\langle \forall\, p \mid p \in \mathcal{P} \rhd \mathit{idle} \wedge \mathit{requestIn}.\textbf{probe} \wedge \mathit{requestIn}.\textbf{current}.\text{proc} = p \leadsto \mathit{current} = p \rangle$$

PROOF
Transition (1) is the only transition that reads from *requestIn*, and it is guarded by *idle* and *requestIn*.**probe**. It is a fair transition, so it must execute at some point if its precondition is stable. By inspection, we can see that no other transition falsifies *idle*, so the precondition must be stable. Therefore, transition (1) executes at some point after the left side of any one of our set of leads-to conditions holds. The postcondition of transition (1) for any given message source is exactly the right side of our leads-to condition for that message source. Therefore, the entire set of leads-to conditions holds. □

THEOREM 5.6 (RESOURCE RELEASE HANDLING PROGRESS)
If a Resource has received more releases from its current client than it has sent tokens to that client, the Resource will eventually become idle.

$$\mathit{busy} \wedge (\mathit{releaseIn} \downarrow_m m.\text{proc} = \mathit{current}).\textbf{cnt} > (\mathcal{O} \downarrow_m m.\text{proc} = \mathit{current}).\textbf{len} \leadsto \mathit{idle}$$

PROOF
From Theorem 5.3, we know that if the left side of our leads-to holds, there is at least one instance of *current* in the *releases* multiset. This means that the precondition for transition (2)

holds, and continues to hold until transition (2) is executed (because transition (2) is the only transition that removes an element from the multiset or negates *busy*). Transition (2) is a fair transition, and weak fairness tells us that it will execute eventually if its precondition holds and is stable. Its postcondition includes *idle*, which proves our leads-to. □

We have now shown all the behavior of the Resource in isolation that we will need to use in our proof of correctness for the composed system.

5.3 The Client Program

---

program Client(resource: **process**)

declare
idle, waiting, busy: **boolean**
tokenIn: **inbox**

always
gone $\triangleq \neg$idle $\wedge \neg$waiting $\wedge \neg$busy

initially
idle = **true** $\wedge$ waiting = **false** $\wedge$ busy = **false**

fair-transition
(1) waiting $\wedge$ tokenIn.**probe** $\longrightarrow$ waiting′ = **false** $\wedge$ busy′ = **true** $\wedge$ tokenIn.**advance**
(2) [] busy $\longrightarrow$ busy′ = **false** $\wedge$ idle′ = **true** $\wedge$ **send**(resource, "releaseIn", $\emptyset$)

unfair-transition
(3) [] idle $\longrightarrow$ idle′ = **false** $\wedge$ waiting′ = **true** $\wedge$ **send**(resource, "requestIn", $\emptyset$)
(4) [] idle $\longrightarrow$ idle′ = **false** $\wedge$ **stop**
(5) [] waiting $\longrightarrow$ waiting′ = **false** $\wedge$ **send**(resource, "releaseIn", $\emptyset$) $\wedge$ **stop**
(6) [] busy $\longrightarrow$ busy′ = **false** $\wedge$ **send**(resource, "releaseIn", $\emptyset$) $\wedge$ **stop**

end

Specification 5.2: The *Client* program, part of the single resource mutual exclusion system

---

The Client program sends requests and releases, and receives tokens; it is considered to have access to whatever resource it needs when it holds a token. We can prove in isolation that the Client program is well-formed, that it never sends two requests without receiving a token in between, and that its message histories and other state variables fulfill certain restrictions that will be useful later in proving the correctness of the composed system.

THEOREM 5.7 (CLIENT WELL-FORMEDNESS)
The Client program is well-formed.

PROOF

We prove that the Client program is well-formed by showing that its always-section, initially-section, and transition-sections are well-formed:

always-section The always-section of the Client program contains one definition, which is a simple Boolean function of three variables. There are no circular definitions, and no undefined variables are referenced. Therefore, the always-section is well-formed.

initially-section The initially-section of the Client program initializes three Boolean variables. There is no recursion, and all three conjuncts of the initially predicate are satisfiable. Therefore, the initially-section is well formed.

transition-sections For each transition, we must demonstrate that its postcondition is always satisfiable if its precondition holds. We do this individually for the six transitions:

- The postcondition of transition (1) is the conjunction of an inbox advance and two assignments to variables, all of which are always satisfiable. Therefore, transition (1)'s postcondition is always satisfiable.
- The postcondition of transition (2) is the conjunction of two assignments to variables and a message send, all of which are always satisfiable. Therefore, transition (2)'s postcondition is always satisfiable.
- The postcondition of transition (3) is the conjunction of two assignments to variables and a message send, all of which are always satisfiable. Therefore, transition (3)'s postcondition is always satisfiable.
- The postcondition of transition (4) is the conjunction of an assignment to a variable and **stop**, both of which are always satisfiable. Therefore, transition (4)'s postcondition is always satisfiable.
- The postcondition of transition (5) is the conjunction of an assignment to a variable, a message send and **stop**, all of which are always satisfiable. Therefore, transition (5)'s postcondition is always satisfiable.
- The postcondition of transition (6) is the conjunction of an assignment to a variable, a message send, and **stop**, all of which are always satisfiable. Therefore, transition (6)'s postcondition is always satisfiable.

Since all sections of the Client program are well-formed, the entire program is well-formed.

□

THEOREM 5.8 (CLIENT STATE TRANSITIONS SAFETY)

Only one of idle, waiting, busy *and* gone *holds for a Client at any given point during its execution. All Client transitions are either from idle to waiting, from waiting to busy, from busy to idle, or from any of these to gone. The gone state is stable.*

$$\begin{aligned}\textbf{invariant}.((\mathit{idle} \Rightarrow \neg \mathit{waiting} \wedge \neg \mathit{busy} \wedge \neg \mathit{gone}) \wedge (\mathit{waiting} \Rightarrow \neg \mathit{busy} \wedge \neg \mathit{gone} \wedge \neg \mathit{idle}) \\ \wedge (\mathit{busy} \Rightarrow \neg \mathit{gone} \wedge \neg \mathit{idle} \wedge \neg \mathit{waiting}) \wedge (\mathit{gone} \Rightarrow \neg \mathit{idle} \wedge \neg \mathit{waiting} \wedge \neg \mathit{busy}))\end{aligned} \tag{1}$$

$$\mathit{idle}\ \textbf{next}\ (\mathit{idle} \vee \mathit{waiting} \vee \mathit{gone}) \tag{2}$$

$$\mathit{waiting}\ \textbf{next}\ (\mathit{waiting} \vee \mathit{busy} \vee \mathit{gone}) \tag{3}$$

$$\mathit{busy}\ \textbf{next}\ (\mathit{busy} \vee \mathit{idle} \vee \mathit{gone}) \tag{4}$$

$$\textbf{stable}.\mathit{gone} \tag{5}$$

PROOF

We prove each of the 5 equations individually:

1. We can eliminate the $(\mathit{gone} \Rightarrow \neg \mathit{idle} \wedge \neg \mathit{waiting} \wedge \neg \mathit{busy})$ conjunct immediately; *gone* is defined as $\neg \mathit{idle} \wedge \neg \mathit{waiting} \wedge \neg \mathit{busy}$, so the implication is a tautology. For the same reason, we can replace $\neg \mathit{gone}$ with **true** in the antecedents of the other three conjuncts. What we have left to prove is that only one of *idle*, *waiting* and *busy* ever holds at any given time. Initially, this is the case, because *idle* is initialized to **true** and *waiting* and *busy* are both initialized to **false**. Transitions (3) and (4), the only transitions enabled when *idle* is **true** and *waiting* and *busy* are **false**, both set *idle* to **false**; transition (3) also sets *waiting* to **true**. Transitions (1) and (5), the only transitions enabled when *waiting* is **true** and *busy* and *idle* are **false**, both set *waiting* to **false**; transition (1) also sets *busy* to **true**. Transitions (2) and (6), the only transitions enabled when *busy* is **true** and *idle* and *waiting* are **false**, both set *busy* to **false**; transition (2) also sets *idle* to **true**. Therefore, only one of *idle*, *waiting* and *busy* ever holds at any given time. This, combined with our elimination of *gone*, proves the invariant.

2. Only two transitions, (3) and (4), are enabled when *idle* holds. Transition (3) sets *idle* to **false**, and *waiting* to **true**. Transition (4) sets *idle* to **false**, establishing *gone* (because *idle*, *waiting* and *busy* are all **false**).

3. Only two transitions, (1) and (5), are enabled when *waiting* holds. Transition (1) sets *waiting* to **false**, and *busy* to **true**. Transition (5) sets *waiting* to **false**, establishing *gone* (because *idle*, *waiting* and *busy* are all **false**).

4. Only two transitions, (2) and (6), are enabled when *busy* holds. Transition (2) sets *busy* to

false, and *idle* to **true**. Transition (6) sets *busy* to **false**, establishing *gone* (because *idle*, *waiting* and *busy* are all **false**).

5. No transitions are enabled when *gone* holds (in addition to the guards all being false, all transitions that cause *gone* to hold also contain a **stop**). Therefore, *gone* is stable, since there is no way for the system to change its value.

THEOREM 5.9 (CLIENT STATE TRANSITIONS PROGRESS)
A Client never stays in the busy state forever.

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

PROOF
Only two transitions, (2) and (6), are enabled when *busy* holds. If transition (6) executes at any point, *busy* is falsified. If transition (6) does not execute, weak fairness ensures that transition (2) will execute at some point, since transition (2) is the only fair transition enabled when *busy* holds. Therefore, *busy* is transient. □

COROLLARY 5.10 (CLIENT BUSY STATE PROGRESSION)
A Client that is in the busy state will at some point in the future be in the idle or gone state.

$$\mathit{busy} \rightsquigarrow \mathit{idle} \vee \mathit{gone}$$

PROOF
This corollary follows immediately from Theorem 5.9 and equation (5) of Theorem 5.8. □

We now prove that specific relationships between Client states and message histories exist in all executions of the Client program. For brevity, we refer to $(\mathcal{O} \downarrow_m m.\text{mbox} = \text{“requestIn”})$, the sequence of messages sent to mailboxes named “requestIn,” as *requestOut*, and $(\mathcal{O} \downarrow_m m.\text{mbox} = \text{“releaseIn”})$, the sequence of messages sent to mailboxes named “releaseIn,” as *releaseOut*. We also define the following predicates that will be used in the proofs of these relationships:

$$P_{idle} \equiv \mathit{idle} \wedge \mathit{requestOut}.\textbf{len} = \mathit{tokenIn}.\textbf{cnt} = \mathit{releaseOut}.\textbf{len}$$

$$P_{waiting} \equiv \mathit{waiting} \wedge \mathit{requestOut}.\textbf{len} - 1 = \mathit{tokenIn}.\textbf{cnt} = \mathit{releaseOut}.\textbf{len}$$

$$P_{busy} \equiv \mathit{requestOut}.\textbf{len} = \mathit{tokenIn}.\textbf{cnt} = \mathit{releaseOut}.\textbf{len} + 1$$

$$P_{gone} \equiv \mathit{requestOut}.\textbf{len} = \mathit{releaseOut}.\textbf{len}$$

LEMMA 5.11 (CLIENT MESSAGE HISTORIES—IDLE/WAITING/GONE)

Assume a Client in the idle state has received exactly as many tokens as it has sent requests and has sent exactly as many releases as it has sent requests. When that Client makes a transition to the waiting state from the idle state, it will have received exactly one fewer token than it has sent requests and sent exactly as many releases as it has received tokens. In addition, when that Client makes a transition to the gone state from the idle state, it will have sent exactly as many releases as it has sent requests.

$$P_{idle} \textbf{ next } P_{idle} \vee P_{waiting} \vee P_{gone}$$

PROOF

Assume P_{idle}, and therefore *idle*, holds in the current state. From Theorem 5.8, we know that if *idle* holds, then exactly one of *idle*, *waiting* or *gone* will hold in the next state.

If *idle* holds in the next state, we know that P_{idle} holds in the next state, because there are no transitions that preserve *idle* while changing the length of an outbox history or the received message index of an inbox.

If *waiting* holds in the next state, we know that the transition that brings about the next state is transition (3), because by inspection it is the only transition enabled when *idle* holds that causes *waiting* to hold as part of its postcondition. P_{idle} tells us that *requestOut*.**len** = *tokenIn*.**len** = *releaseOut*.**len** in the current state. Transition (3) sends a message to an inbox named "requestIn," increasing *requestOut*.**len** by 1, and does not change any other mailbox states. Therefore, *requestOut*.**len** $-$ 1 = *tokenIn*.**cnt** = *releaseOut*.**len** must hold in the next state. The conjunction of this and *waiting* is exactly $P_{waiting}$.

If *gone* holds in the next state, we know that the transition that brings about the next state is transition (4), because by inspection it is the only transition enabled when *idle* holds that causes *gone* to hold as part of its postcondition. P_{idle} tells us that *requestOut*.**len** = *releaseOut*.**len** in the current state. Transition (4) does not change any mailbox states. Therefore, *requestOut*.**len** = *releaseOut*.**len** must hold in the next state. The conjunction of this and *gone* is exactly P_{gone}.

□

LEMMA 5.12 (CLIENT MESSAGE HISTORIES—WAITING/BUSY/GONE)

Assume a Client in the waiting state has received exactly one fewer token than it has sent requests and sent exactly as many releases as it has received tokens. When that Client makes a transition to the busy state from the waiting state, it will have received exactly as many tokens as it has sent requests and sent exactly one fewer release than it has received tokens. In addition, when that Client makes a transition to the gone state from the waiting state, it will have sent exactly

as many releases as it has sent requests.

$$P_{waiting} \textbf{ next } P_{waiting} \vee P_{busy} \vee P_{gone}$$

PROOF

Assume $P_{waiting}$, and therefore *waiting*, holds in the current state. From Theorem 5.8, we know that if *waiting* holds, then exactly one of *waiting*, *busy* or *gone* will hold in the next state.

If *waiting* holds in the next state, we know that $P_{waiting}$ holds in the next state, because there are no transitions that preserve *waiting* while changing the length of an outbox history or the received message index of an inbox.

If *busy* holds in the next state, we know that the transition that brings about the next state is transition (1), because by inspection it is the only transition enabled when *waiting* holds that causes *busy* to hold as part of its postcondition. $P_{waiting}$ tells us that *requestOut*.**len** $-$ 1 $=$ *tokenIn*.**cnt** $=$ *releaseOut*.**len** in the current state. Transition (1) advances *tokenIn*, increasing *tokenIn*.**cnt** by 1, and does not change any other mailbox states. Therefore, *requestOut*.**len** $=$ *tokenIn*.**cnt** $=$ *releaseOut*.**len** $+$ 1 must hold in the next state. The conjunction of this and *busy* is exactly P_{busy}.

If *gone* holds in the next state, we know that the transition that brings about the next state is transition (5), because by inspection it is the only transition enabled when *waiting* holds that causes *gone* to hold as part of its postcondition. $P_{waiting}$ tells us that *requestOut*.**len** $-$ 1 $=$ *tokenIn*.**cnt** $=$ *releaseOut*.**len** in the current state. Transition (5) sends a message to an inbox named "releaseIn," increasing *releaseOut*.**len** by 1, and does not change any other mailbox states. Therefore, *requestOut*.**len** $=$ *releaseOut*.**len** must hold in the next state. The conjunction of this and *gone* is exactly P_{gone}. □

LEMMA 5.13 (CLIENT MESSAGE HISTORIES—BUSY/IDLE/GONE)

Assume a Client in the busy state has received exactly as many tokens as it has sent requests and sent exactly one fewer release than it has received tokens. When that Client makes a transition to the idle state from the busy state, it will have received exactly as many tokens as it has sent requests and sent exactly as many releases as it has received tokens. In addition, when that Client makes a transition to the gone state from the busy state, it will have sent exactly as many releases as it has sent requests.

$$P_{busy} \textbf{ next } P_{busy} \vee P_{waiting} \vee P_{gone}$$

PROOF

Assume P_{busy}, and therefore *busy*, holds in the current state. From Theorem 5.8, we know that if *busy* holds, then exactly one of *busy*, *idle* or *gone* will hold in the next state.

If *busy* holds in the next state, we know that P_{busy} holds in the next state, because there are no transitions that preserve *busy* while changing the length of an outbox history or the received message index of an inbox.

If *idle* holds in the next state, we know that the transition that brings about the next state is transition (2), because by inspection it is the only transition enabled when *busy* holds that causes *idle* to hold as part of its postcondition. P_{busy} tells us that *requestOut*.**len** = *tokenIn*.**cnt** = *releaseOut*.**len** + 1 in the current state. Transition (2) sends a message to an inbox named "releaseIn," increasing *releaseOut*.**len** by 1, and does not change any other mailbox states. Therefore, *requestOut*.**len** = *tokenIn*.**cnt** = *releaseOut*.**len** must hold in the next state. The conjunction of this and *idle* is exactly P_{idle}.

If *gone* holds in the next state, we know that the transition that brings about the next state is transition (6), because by inspection it is the only transition enabled when *busy* holds that causes *gone* to hold as part of its postcondition. P_{busy} tells us that *requestOut*.**len** = *tokenIn*.**cnt** = *releaseOut*.**len** + 1 in the current state. Transition (6) sends a message to an inbox named "releaseIn," increasing *releaseOut*.**len** by 1, and does not change any other mailbox states. Therefore, *requestOut*.**len** = *releaseOut*.**len** must hold in the next state. The conjunction of this and *gone* is exactly P_{gone}. □

THEOREM 5.14 (CLIENT STATES AND MESSAGE HISTORIES)

A Client in the idle state has received exactly as many tokens as it has sent requests and has sent exactly as many releases as it has received tokens. A Client in the waiting state has received exactly one fewer token than it has sent requests and sent exactly as many releases as it has received tokens. A Client in the busy state has received exactly as many tokens as it has sent requests and sent exactly one fewer release than it has received tokens. A Client in the gone state has sent exactly as many releases as it has sent requests.

$$\textbf{invariant}.((idle \Rightarrow P_{idle}) \wedge (waiting \Rightarrow P_{waiting}) \wedge (busy \Rightarrow P_{busy}) \wedge (gone \Rightarrow P_{gone}))$$

PROOF

Initially, the Client is in a state where P_{idle} holds: *idle* is initialized to **true**, and all mailboxes are initially empty. By induction, Lemmas 5.11, 5.12 and 5.13 tell us that this invariant holds for all states if it holds for the initial state, since the combination of those theorems shows that all transitions from state X where P_X holds are to state Y where P_Y holds, where X and Y can each be one of idle, busy, waiting or gone, subject to the permissible state transitions shown

in Theorem 5.8. □

COROLLARY 5.15 (CLIENT TOKENS/RELEASES SAFETY)

The difference between the number of tokens a Client has received and the number of releases it has sent is always between -1 and 1, inclusive.

$$\textbf{invariant}.(-1 \leq \mathit{tokenIn}.\textbf{cnt} - \mathit{releaseOut}.\textbf{len} \leq 1)$$

PROOF

This corollary follows immediately from Theorem 5.14. □

COROLLARY 5.16 (CLIENT TOKENS/RELEASES PROGRESS)

If a Client has received more tokens than it has sent releases, at some point it will have sent exactly as many releases as it has received tokens.

$$\mathit{tokenIn}.\textbf{cnt} > \mathit{releaseOut}.\textbf{len} \leadsto \mathit{tokenIn}.\textbf{cnt} = \mathit{releaseOut}.\textbf{len}$$

PROOF

This corollary follows immediately from Corollary 5.10 and Theorem 5.14. □

We have now shown all the behavior of the Client in isolation that we will need to use in our proof of correctness for the composed system.

5.4 The Composed System

The composed system contains the Resource program (Specification 5.1), the Client program (Specification 5.2), and a small initial program called Generator that takes care of creating the Resource and the Clients. We first prove some properties about this Generator program, and then prove the correctness of the composed system.

5.4.1 The Generator Program

The Generator program is responsible for creating a Resource process in the system, as well as for creating Client processes for that Resource. We prove that the Generator is well-formed, and that its reference to the Resource is stable (that is, that its *resource* state variable always refers to the same process in the system).

THEOREM 5.17 (GENERATOR WELL-FORMEDNESS)

The Generator program is well-formed.

```
system SingleResourceMutualExclusion

  initial-program Generator

    declare
      resource: process

    initially
      resource = new Resource

    fair-transition
      (1)  p: p′ = new Client(resource)

  end

  program Resource

  program Client(resource: process)

end
```

Specification 5.3: The *SingleResourceMutualExclusion* system

PROOF

We prove that the Generator program is well-formed by showing that its always-section, initially-section, and transition-sections are well-formed:

always-section The Generator program has no always-section, so we need not show that its always-section is well-formed.

initially-section The initially-section of the Generator program instantiates a Resource process. This is not an infinitely recursive instantiation, since the Resource process is well-formed (as shown in Theorem 5.1). Moreover, it is satisfiable. Therefore, the Generator program's initially-section is well-formed.

transition-sections For each transition, we must demonstrate that its postcondition is always satisfiable if its precondition holds. There is only one transition in the Generator program, which is an instantiation of a Client process. This is always satisfiable (since the Client process is well-formed, as shown in Theorem 5.7). Therefore, the Generator program's transition-sections are well-formed.

Since all sections of the Generator program are well-formed, the entire program is well-formed. □

THEOREM 5.18 (GENERATOR RESOURCE REFERENCE STABILITY)
Once a process is instantiated and a reference to that process is assigned to resource *in a Generator process,* resource *refers to that process forever:*

$$\langle \forall p \rhd \textbf{stable}.(resource = p) \rangle$$

PROOF
Initially, *resource* is assigned to be a reference to a new instance of the Resource program. No transition in the Resource program modifies the *resource* reference. Therefore, *resource* refers to the same instance of the Resource program forever. □

We have now shown all the behavior of the Generator program in isolation that we will need to use in our proof of correctness for the composed system.

5.4.2 Proof of Correctness

We now show that the entire system implements a single resource mutual exclusion algorithm, by proving several theorems about the system using the properties we have already proven about the Generator, Resource and Client programs in isolation.

THEOREM 5.19 (GENERATOR AND RESOURCE UNIQUENESS/STABILITY)
There is exactly one Generator process and exactly one Resource process in the SingleResource-MutualExclusion system, and these processes never stop.

PROOF
We need to prove that a single Generator process is created in every possible execution of the system, that a single Resource process is created in every possible execution of the system, that it is impossible for any other Generator or Resource processes to be created, and that these processes never stop.

Generator is the initial program of the system, so a single instance of Generator is created at system initialization time. Theorem 5.17 shows that the Generator program is well-formed, and by inspection it does not contain a **stop** statement. Therefore, in accordance with Dynamic UNITY execution semantics, it runs forever. Also by inspection, no program contains a statement that instantiates a Generator process. Therefore, no other Generator process is ever instantiated during the execution of the system.

A single Resource process is instantiated by the initially-section of the Generator program. By inspection, there are no other statements in any program that instantiate a Resource process. We have already shown that exactly one Generator process is created. Therefore, exactly one Resource process is created during the Generator process's construction. Theorem 5.1

shows that the Resource program is well-formed, and by inspection it does not contain a **stop** statement. Therefore, in accordance with Dynamic UNITY execution semantics, it runs forever. □

THEOREM 5.20 (CLIENT RESOURCE REFERENCE UNIQUENESS)
All Client processes in the SingleResourceMutualExclusion system have references to the same Resource process.

$$\textbf{invariant}.(\langle \forall r \mid r \in \mathcal{P} \wedge r = \mathit{resource} \triangleright \langle \forall c, q \mid c \in \mathcal{P} \wedge c = \text{Client}(q) \triangleright q = r \rangle \rangle)$$

PROOF
By inspection, the only statement in any program that instantiates a Client process is part of transition (1) of the Generator program. Theorem 5.19 shows that there is exactly one Resource process in the system, so the quantification always ranges over exactly one Resource process, and also shows that the Resource process is created by the one Generator process in the system. Theorem 5.18 shows that the Generator's *resource* variable always refers to this Resource process. Therefore, all Client processes created by the Generator program have a reference to the same Resource process as their *resource* parameter. □

In the following theorems, we refer to the single Resource process in the SingleResource-MutualExclusion system as R. For brevity, we refer to $(\mathcal{O}^p \downarrow_m m.\text{mbox} = \text{“requestIn”})$, the sequence of messages sent to mailboxes named “requestIn” by process p, as $p.\mathit{requestOut}$, and $(\mathcal{O}^p \downarrow_m m.\text{mbox} = \text{“releaseIn”})$, the sequence of messages sent to mailboxes named “releaseIn” by process p, as $p.\mathit{releaseOut}$.

THEOREM 5.21 (MESSAGING CONNECTIONS)
For every Client process c*, the message histories of* R.requestIn *filtered by* proc = c *and* R.release-In *filtered by* proc = c *follow the message histories of* c.requestOut *and* c.releaseOut *respectively. For every Client process* c*, the message history of* c.tokenIn *follows the message history of* $\mathcal{O}^{\text{R}}$ *filtered by* proc = c.

$$\begin{aligned}\textbf{invariant}.(\langle \forall c \mid & c \in \mathcal{P} \wedge c = \text{Client}(R) \triangleright \\ & (R.\mathit{requestIn} \downarrow_m m.\text{proc} = c).\text{msg } \textbf{follows } c.\mathit{requestOut}.\text{msg} \wedge \\ & (R.\mathit{releaseIn} \downarrow_m m.\text{proc} = c).\text{msg } \textbf{follows } c.\mathit{releaseOut}.\text{msg} \wedge \\ & c.\mathit{tokenIn}.\text{msg } \textbf{follows } (\mathcal{O}^R \downarrow_m m.\text{proc} = c).\text{msg})\end{aligned}$$

PROOF
By inspection, all messages sent by a Client are sent to the inbox named “requestIn” associated with the Resource process to which the Client has a reference. Theorem 5.20 says that this

Resource process is R, for all Client processes. Also by inspection, the "requestIn" inbox of R is always $R.requestIn$. It immediately follows from the Channel Theorem that $R.requestIn$ filtered by proc = c follows $c.requestOut$. By symmetry, the same is true for the release outbox/inbox pair.

By inspection, all messages sent by R are sent to the inbox named "tokenIn" associated with the destination process. If the destination process is a Client instantiation, the "tokenIn" inbox is always the *tokenIn* variable of that process; if the destination process is not a Client instantiation, it has no bearing on this theorem. It immediately follows from the channel theorem (Theorem 3.11) that, for all Client processes, the *tokenIn* message history follows R's outgoing message sequence filtered by proc = c. □

We now prove system safety, by showing that there is exactly one live token in the system at all times. The number of live tokens in the system is defined as the sum of the number of tokens held by the Resource, the number of tokens held by Clients, the number of live tokens in transit from Clients to the Resource, and the number of live tokens in transit from the Resource to Clients. A live token in transit from a Client to a Resource is a release message whose corresponding request that has been handled by the Resource. If there are two release messages in transit from a Client to the Resource, there is only one live token in transit from that Client to the Resource (because, as we will see, message histories dictate that there also be an unhandled request message in transit from that Client to the Resource). A live token in transit from a Resource to a Client is a token message whose destination Client is not in the gone state. Since a Client in the gone state never receives any messages, a token message sent to such a Client will never be received and therefore does not play a further role in the system.

For use in our safety proof, we define the following quantities (some of which are parameterized by a Client c) that represent the numbers of tokens in transit in the system in accordance with the above definitions. R_{out} is the number of live tokens in transit from the Resource that are still in the Resource's outbox; $C_{in}(c)$ is the number of live tokens in transit from the Resource that are in Client c's *tokenIn* inbox; $R_{in}(c)$ is the number of tokens in transit from Client c to the Resource that are either in the Resource's *releaseIn* inbox or in its *releases* multiset; and $C_{out}(c)$ is the number of tokens in transit from Client c to the Resource that are still in the Client's outbox. For brevity, from this point forward, we denote the set of clients in the system (that is, the set of all processes of type Client(R)) by CS:

$$R_{out} \equiv \langle \#i \mid \mathcal{O}^R.\mathbf{cnt} \leq i < \mathcal{O}^R.\mathbf{len} \rhd \neg \mathcal{O}^R[i].\text{proc}.gone \rangle$$

$$C_{in}(c) \equiv \langle \Sigma d \mid d = c \wedge \neg d.gone \rhd d.tokenIn.\mathbf{len} - d.tokenIn.\mathbf{cnt} \rangle$$

$$R_{in}(c \neq R.current) \equiv$$
$$\langle \#i \mid R.releaseIn.\mathbf{cnt} \leq i < R.releaseIn.\mathbf{len} \rhd R.releaseIn[i].\text{proc} = c \rangle -$$
$$\langle \#i \mid R.requestIn.\mathbf{cnt} \leq i < R.requestIn.\mathbf{len} \rhd R.requestIn[i].\text{proc} = c \rangle +$$
$$\langle \#p \mid p \in R.releases \rhd p = c \rangle$$

$$R_{in}(c = R.current) \equiv$$
$$\langle \#i \mid R.releaseIn.\mathbf{cnt} \leq i < R.releaseIn.\mathbf{len} \rhd R.releaseIn[i].\text{proc} = c \rangle +$$
$$\langle \#p \mid p \in R.releases \rhd p = c \rangle$$

$$C_{out}(c \neq R.current) \equiv$$
$$(c.releaseOut.\mathbf{len} - c.releaseOut.\mathbf{cnt}) - (c.requestOut.\mathbf{len} - c.requestOut.\mathbf{cnt})$$

$$C_{out}(c = R.current) \equiv c.releaseOut.\mathbf{len} - c.releaseOut.\mathbf{cnt}$$

The actual numbers of live tokens in the system held by the Resource, the Clients, and in transit, are defined as follows (note that we do not allow there to be negative live tokens in the system, and negative tokens in transit are not included in the sums):

$$LT_{resource} \equiv \langle \#r \mid r = R \rhd R.idle \rangle$$

$$LT_{clients} \equiv \langle \#c \mid c \in CS \rhd C.busy \rangle$$

$$LT_{transit} \equiv R_{out} + \langle \Sigma c \mid c \in CS \wedge c \neq R.current \rhd C_{in}(c) + \max(C_{out}(c) + R_{in}(c), 0) \rangle +$$
$$C_{in}(R.current) + \langle \#c \mid c = R.current \rhd 1 \leq R_{in}(c) + C_{out}(c) \leq 2 \rangle$$

We now use these equations to define the following predicates, which describe the possible states of the tokens in the system:

$$NOTBUSY \equiv \langle \forall c \mid c \in CS \rhd \neg c.busy \rangle$$
$$BUSY(c) \equiv c.busy \wedge \langle \forall d \mid d \in CS \wedge d \neq c \rhd \neg d.busy \rangle$$

$$P_{resource} \equiv \langle \forall c \mid c \in CS \rhd -1 \leq R_{in}(c) + C_{out}(c) \leq 0 \rangle \wedge$$
$$R_{out} = 0 \wedge \langle \forall c \mid c \in CS \rhd C_{in}(c) = 0 \rangle \wedge R.idle \wedge NOTBUSY$$

$$P_{toclient} \equiv \langle \forall c \mid c \in CS \rhd -1 \leq R_{in}(c) + C_{out}(c) \leq 0 \rangle \wedge R_{out} + C_{in}(R.current) = 1 \wedge$$
$$R_{out} + \langle \Sigma c \mid c \in CS \rhd C_{in}(c) \rangle = 1 \wedge R.busy \wedge NOTBUSY$$

$$P_{client} \equiv \langle \forall c \mid c \in CS \rhd -1 \leq R_{in}(c) + C_{out}(c) \leq 0 \rangle \wedge R_{out} = 0 \wedge$$
$$\langle \forall c \mid c \in CS \rhd C_{in}(c) = 0 \rangle \wedge R.busy \wedge BUSY(R.current)$$

$$P_{toresource} \equiv 1 \leq R_{in}(R.current) + C_{out}(R.current) \leq 2 \wedge$$
$$\langle \forall c \mid c \in CS \wedge c \neq R.current \rhd -1 \leq R_{in}(c) + C_{out}(c) \leq 0 \rangle \wedge$$
$$R_{out} = 0 \wedge \langle \forall c \mid c \in CS \rhd C_{in}(c) = 0 \rangle \wedge R.busy \wedge NOTBUSY$$

Lemma 5.22 (System State Transitions—Resource)
If the system is in the state where the Resource holds a live token ($P_{toresource}$), the next state will be either the same, the state where a live token is in transit to the Resource's current Client ($P_{toclient}$), or the state where a live token is in transit to the Resource from the Resource's current Client ($P_{toresource}$).

$$P_{resource} \textbf{ next } P_{resource} \vee P_{toclient} \vee P_{toresource}$$

Proof

Assume $P_{resource}$ holds in the current state. We examine all possible state transitions in the system to determine the possible next states.

Transition (1) of the Resource program may be enabled, since $R.idle$ holds in the current state. If transition (1) is enabled, then there is a message waiting on *requestIn* from a particular Client c. Execution of the transition reads this message, falsifies $R.idle$ by setting $R.current$ to c, and sends a message to c's *tokenIn* inbox. If c is not in the gone state, this transition establishes $P_{toclient}$ in the next state, as follows: for all clients d, $d \neq c$, $-1 \leq R_{in}(d) + C_{out}(d) \leq 0$ still holds because transition (1) hasn't changed it. $-1 \leq R_{in}(c) + C_{out}(c) \leq 0$ holds, because the definitions of these terms for $c = R.current$ combined with the message histories for a Client that is not in the gone state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21) imply that both $R_{in}(c)$ and $C_{out}(c)$ are equal to 0. $R_{out} + C_{in}(R.current) = 1$ holds, because this sum was previously 0, and R_{out} is now 1. $R_{out} + \langle \Sigma c \mid c \in CS \rhd C_{in}(c) \rangle = 1$ holds, because this sum was previously 0 and R_{out} is now 1. $R.busy$ holds because it is part of the postcondition ($R.current$ is assigned a value), and $NOTBUSY$ holds because the transition does not change any client states. The conjunction of these is exactly $P_{toclient}$. If c is in the gone state, this transition establishes $P_{toresource}$ in the next state, as follows: $R_{in}(c) + C_{out}(c) = 1$ holds, because the definitions of these terms for $c = R.current$ combined with the message histories for a Client that is not in the gone

state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21) imply that exactly one of $R_{in}(c)$ and $C_{out}(c)$ is equal to 1. For all clients d, $d \neq c$, $-1 \leq R_{in}(d) + C_{out}(d) \leq 0$ holds because transition (1) hasn't changed it. $R_{out} = 0$ holds because the message that transition (1) sends doesn't contribute to R_{out} (its destination is in the gone state), so R_{out} is left unchanged. $\langle \forall c \mid c \in CS \rhd C_{in}(c) = 0 \rangle$ holds because it held in the previous state and is unchanged by transition (1). $R.busy$ holds because it is part of the postcondition, and *NOTBUSY* holds because the transition does not change any client states. The conjunction of these is exactly $P_{toresource}$.

Transition (2) of the Resource program is not enabled, because $R.busy$ does not hold.

Transition (3) of the Resource program may be enabled. If it is executed, it reads a message from $R.releaseIn$ and places an entry in $R.releases$. This does not change $R_{in}(c)$ for any c, because it is subtracting 1 from the number of unread messages from c while adding 1 to the number of appearances of c in the set. Therefore, the system state after transition (3) is still $P_{resource}$.

Transition (1) of the Client program is not enabled for any Client c, because $C_{in}(c) = 0$ for all c.

Transitions (2) and (6) of the Client program are not enabled for any Client c, because *NOTBUSY* holds.

Transition (3) of the Client program is enabled for Client c only if $c.idle$ holds. It sends a message to $R.requestIn$, decrementing $C_{out}(c)$ by 1. Because $c.idle$ holds, and the Resource is idle, we know that $R_{in}(c) + C_{out}(c) = 0$. This follows from the message histories for a Client that is in the idle state state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21). Therefore, the system state after transition (3) is still $P_{resource}$, but with $R_{in}(c) + C_{out}(c) = -1$.

Transition (4) of the Client program does not send a message or cause a Client to enter the busy state, so it does not falsify $P_{resource}$.

Transition (5) of the Client program is enabled for Client c only if $c.waiting$ holds. It sends a message to $R.releaseIn$, incrementing $C_{out}(c)$ by 1. Because $c.waiting$ holds, and the Resource is idle, we know that $R_{in}(c) + C_{out}(c) = -1$. This follows from the message histories for a Client that is in the idle state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21). Therefore, the system state after transition (3) is still $P_{resource}$, but with $R_{in}(c) + C_{out}(c) = 0$.

If an outgoing message from c is delivered, $P_{resource}$ is maintained, since delivery of the message increments or decrements $C_{out}(c)$ by 1 and correspondingly decrements or increments $R_{in}(c)$ by 1.

The transitions of the Generator do not affect the next property, since they do not cause any messages to be sent or received and all Clients created by the Generator are initially idle.

Therefore, all transitions in the system maintain the next property. □

LEMMA 5.23 (SYSTEM STATE TRANSITIONS—TO CLIENT)

If the system is in the state where a live token is in transit to the Resource's current Client ($\mathrm{P}_{\mathrm{toclient}}$*), the next state will be either the same, the state where the Resource's current Client holds a live token (*$\mathrm{P}_{\mathrm{client}}$*), or the state where a live token is in transit to the Resource from the Resource's current Client (*$\mathrm{P}_{\mathrm{toresource}}$*).*

$$P_{toclient} \textbf{ next } P_{toclient} \vee P_{client} \vee P_{toresource}$$

PROOF

Assume $P_{toclient}$ holds in the current state. We examine all possible state transitions in the system to determine the possible next states.

Transition (1) of the Resource program is not enabled, because $R.busy$ holds.

Transition (2) of the Resource program is not enabled, because we must have $R_{in}(R.current) = 0$ to satisfy the $R_{in}(R.current) + C_{out}(R.current) \leq 0$ in $P_{toclient}$, and this means that there are no instances of $R.current$ in the $releases$ multiset.

Transition (3) of the Resource program may be enabled. If it is executed, it reads a message from $R.releaseIn$ and places an entry in $R.releases$. This does not change $R_{in}(c)$ for any c, because it is subtracting 1 from the number of unread messages from c while adding 1 to the number of appearances of c in the set. Therefore, the system state after transition (3) is still $P_{toclient}$.

A message may be delivered from the Resource's outbox to its destination inbox. Assume the delivered message is destined for Client c. If c is in the gone state, neither R_{out} nor $C_{in}(c)$ is changed (by definition); if c is not in the gone state, R_{out} is decremented (to 0) and $C_{in}(c)$ is incremented (to 1), maintaining $P_{toclient}$.

Transition (1) of the Client program may be enabled for $c = R.current$, but will not be enabled for any other Client because $C_{in}(c \neq R.current) = 0$. This transition reads a message from $tokenIn$, which decrements $C_{in}(c)$ by 1 and makes $R_{out} = 0$ and $C_{in}(c) = 0$. After the transition, therefore, we have $R_{out} = 0$ and $\langle \forall c \mid c \in CS \rhd C_{in}(c) = 0 \rangle$. The transition does not change $c.requestOut$ or $c.releaseOut$ or set the state of c to gone, so after its execution $\langle \forall c \mid c \in CS \rhd -1 \leq R_{in}(c) + C_{out}(c) \leq 0 \rangle$ still holds. $R.busy$ still holds, because the transition does not affect it. Client c is in the busy state after the transition, so $BUSY(c)$ holds because the states of the other Clients, all of which were idle, do not change. The conjunction of these is exactly P_{client}.

Transitions (2) and (6) of the Client program are not enabled for any Client c, because *NOTBUSY* holds.

Transition (3) of the Client program is enabled for Client c only if $c.idle$ holds. It sends a message to $R.requestIn$, decrementing $C_{out}(c)$ by 1. Because $c.idle$ holds, and the Resource is busy, we know that $R_{in}(c) + C_{out}(c) = 0$ for all $c \neq R.current$. This follows from the message histories for a Client that is in the idle state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21). We also know, from the same message histories and connections, that $R.current$ must be in either the waiting or gone state, so its transition (3) is not enabled. Therefore, the system state after transition (3) is still $P_{toclient}$, but with $R_{in}(c) + C_{out}(c) = -1$.

Transition (4) of the Client program does not send a message or cause a Client to enter the busy state, so it does not falsify $P_{toclient}$.

Transition (5) of the Client program is enabled for Client c only if $c.waiting$ holds. It sends a message to $R.releaseIn$, incrementing $C_{out}(c)$ by 1. Because $c.waiting$ holds, and the Resource is busy, we know that $R_{in}(c) + C_{out}(c) = -1$ for all $c \neq R.current$. This follows from the message histories for a Client that is in the idle state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21). Therefore, the system state after transition (5) is still $P_{toclient}$, but with $R_{in}(c) + C_{out}(c) = 0$, if $c \neq R.current$. If $c = R.current$, we know from the same message histories and connections that $R_{in}(c) + C_{out}(c) = 0$. The sending of the message to $R.releaseOut$ increases this sum to 1. The transition also establishes $R_{out} = 0 \wedge \langle \forall c \mid c \in CS \rhd C_{in}(c) = 0 \rangle$, because it causes the Client to transition to the gone state, making that sum equal to 0 for all Clients (it was previously 0 for all Clients other than $R.current$). The conjunction of these and the parts of $P_{toclient}$ that are not changed by the transition is exactly $P_{toresource}$.

One or more messages from any Client except $R.current$ to the Resource may be delivered. This follows from the message histories for a Client that is in the waiting state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21). If this occurs message is delivered, $P_{toclient}$ is maintained, since it increments or decrements $C_{out}(c)$ and correspondingly decrements or increments $R_{in}(c)$.

The transitions of the Generator do not affect the next property, since they do not cause any messages to be sent or received and all Clients created by the Generator are initially idle.

Therefore, all transitions in the system maintain the next property. □

LEMMA 5.24 (SYSTEM STATE TRANSITIONS—CLIENT)

If the system is in a state where the Resource's current Client holds a live token (P_{client}), the next state will be either the same or the state where a live token is in transit to the Resource from the

Resource's current Client ($\mathrm{P}_{\mathrm{toresource}}$).

$$P_{client} \textbf{ next } P_{client} \vee P_{toresource}$$

PROOF

Assume P_{client} holds in the current state. We examine all possible state transitions in the system to determine the possible next states.

Transition (1) of the Resource program is not enabled, because $R.busy$ holds.

Transition (2) of the Resource program is not enabled, because we must have $R_{in}(R.current) = 0$ to satisfy the $R_{in}(R.current) + C_{out}(R.current) \leq 0$ in P_{client}, and this means that there are no instances of $R.current$ in the *releases* multiset.

Transition (3) of the Resource program may be enabled. If it is executed, it reads a message from $R.releaseIn$ and places an entry in $R.releases$. This does not change $R_{in}(c)$ for any c. If $c \neq R.current$, it subtracts 1 from the number of unread messages from c while adding 1 to the number of appearances of c in the set; if $c = R.current$, it is not possible that there was a message from c on *releaseIn*, because c is in the busy state. This follows from the message histories for a Client that is in the busy state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21). Therefore, the system state after transition (3) is still P_{client}.

Transition (1) of the Client program is not enabled for any Client, because $\langle \forall c \mid c \in CS \rhd C_{in}(c) = 0 \rangle$ holds.

Transition (2) of the Client program is enabled for $R.current$, but not for any other Client, since $BUSY(R.current)$ holds. Execution of this transition falsifies $R.current.busy$, established $R.current.idle$, and sends a message to from $R.current$ to $R.releaseIn$. We know, from the message histories for a Client that is in the busy state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21), that $R_{in}(R.current) + C_{out}(R.current) = 0$ holds before execution of the transition; the sending of the message to $R.releaseIn$ increases this sum to 1. The transition also establishes $NOTBUSY$, because it puts $R.current$ into the idle state. The conjunction of these and the parts of P_{client} that are unchanged by execution of transition (2) is exactly $P_{toresource}$.

Transition (3) of the Client program is enabled for Client c only if $c.idle$ holds. It sends a message to $R.requestIn$, decrementing $C_{out}(c)$ by 1. Because $c.idle$ holds, and the Resource is busy, we know that $R_{in}(c) + C_{out}(c) = 0$ for all $c \neq R.current$. This follows from the message histories for a Client that is in the idle state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21). We also know that $R.current$ is in the busy state, since $BUSY(R.current)$ holds, so its transition (3) is not enabled.

Therefore, the system state after transition (3) is still P_{client}, but with $R_{in}(c) + C_{out}(c) = -1$.

Transition (4) of the Client program is enabled only for Clients in the idle state (and is therefore disabled for $R.current$). It does not send a message or cause a Client to enter the busy state, so it does not falsify P_{client}.

Transition (5) of the Client program is enabled for Client c only if $c.waiting$ holds. It sends a message to $R.releaseIn$, incrementing $C_{out}(c)$ by 1. Because $c.waiting$ holds, and the Resource is busy, we know that $R_{in}(c) + C_{out}(c) = -1$ for all $c \neq R.current$. This follows from the message histories for a Client that is in the idle state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21). Therefore, the system state after transition (5) is still P_{client}, but with $R_{in}(c) + C_{out}(c) = 0$, if $c \neq R.current$. We know that $R.current$ is in the busy state, since $BUSY(R.current)$ holds. Therefore, its transition (5) is not enabled.

One or more messages from any Client except $R.current$ to the Resource may be delivered. This follows from the message histories for a Client that is in the waiting state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21). If this occurs message is delivered, $P_{toclient}$ is maintained, since it increments or decrements $C_{out}(c)$ and correspondingly decrements or increments $R_{in}(c)$.

The transitions of the Generator do not affect the next property, since they do not cause any messages to be sent or received and all Clients created by the Generator are initially idle.

Therefore, all transitions in the system maintain the next property. □

LEMMA 5.25 (SYSTEM STATE TRANSITIONS—TO RESOURCE)

If the system is in a state where a live token is in transit to the Resource from the Resource's current Client ($P_{\text{toresource}}$), the next state will be either the same or the state where the Resource holds a live token (P_{resource}).

$$P_{toresource} \textbf{ next } P_{toresource} \vee P_{resource}$$

PROOF

Assume $P_{toresource}$ holds in the current state. We examine all possible state transitions in the system to determine the possible next states.

Transition (1) of the Resource program is not enabled, because $R.busy$ holds.

Transition (2) of the Resource program is enabled only if $\langle \#p \mid p \in releases \rhd p = R.current \rangle \geq 1$. Execution of transition (2) removes an instance of $R.current$ from $releases$, and sets $R.current'$ to $\perp$. This establishes $\langle \forall c \mid c \in CS \rhd -1 \leq R_{in}(c) + C_{out}(c) \leq 0 \rangle$, as follows: for all $c \neq R.current$, the inequality already held, and for $c = R.current$ we had $1 \leq R_{in}(c) + C_{out}(c) \leq 2$. This means that there were either 1 or 2 release messages in transit from $R.current$ to the Re-

source, and after execution of transition (2) there are 0 or 1 release messages in transit from that Client. From the message histories for a Client that is not in the busy state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21), we know that the maximum number of request messages in transit from a non-busy Client to the Resource is 1, and that if 2 release messages are in transit from a non-busy Client, then 1 request message is in transit from that Client as well. The definitions of $R_{in}(c)$ and $C_{out}(c)$ for $c = R.current$ differ from those for $c \neq R.current$ only in that the former subtract the number of request messages from the latter. Therefore, we have $-1 \leq R_{in}(c) + C_{out}(c) \leq 0$ for all c after execution of transition (2). Transition (2) also establishes $R.idle$. The conjunction of these and the parts of $P_{toresource}$ that are unchanged by execution of transition (2) is exactly $P_{resource}$.

Transition (3) of the Resource program may be enabled. If it is executed, it reads a message from $R.releaseIn$ and places an entry in $R.releases$. This does not change $R_{in}(c)$ for any c. If $c \neq R.current$, it subtracts 1 from the number of unread messages from c while adding 1 to the number of appearances of c in the set; if $c = R.current$, it is not possible that there was a message from c on $releaseIn$, because c is in the busy state. This follows from the message histories for a Client that is in the busy state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21). Therefore, the system state after transition (3) is still $P_{toresource}$.

Transition (1) of the Client program is not enabled for any Client, because $\langle \forall c \mid c \in CS \rhd C_{in}(c) = 0 \rangle$ holds.

Transitions (2) and (6) of the Client program are not enabled for any Client c, because $NOTBUSY$ holds.

Transition (3) of the Client program is enabled for Client c only if $c.idle$ holds. It sends a message to $R.requestIn$, decrementing $C_{out}(c)$ by 1. Because $c.idle$ holds, and the Resource is idle, we know that $R_{in}(c) + C_{out}(c) = 0$. This follows from the message histories for a Client that is in the idle state state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21). Therefore, the system state after transition (3) is still $P_{toresource}$, but with $R_{in}(c) + C_{out}(c) = -1$.

Transition (4) of the Client program does not send a message or cause a Client to enter the busy state, so it does not falsify $P_{resource}$.

Transition (5) of the Client program is enabled for Client c only if $c.waiting$ holds. It sends a message to $R.releaseIn$, incrementing $C_{out}(c)$ by 1. Because $c.waiting$ holds, and the Resource is idle, we know that $R_{in}(c) + C_{out}(c) = -1$. This follows from the message histories for a Client that is in the idle state (Theorem 5.14), the message histories for the Resource (Theorem 5.3) and the messaging connections (Theorem 5.21). Therefore, the system state after transition (3)

is still $P_{toresource}$, but with $R_{in}(c) + C_{out}(c) = 0$.

One or more messages to the Resource may be delivered. If one is, $P_{toresource}$ is maintained, since the message delivery increments or decrements $C_{out}(c)$ by 1 and correspondingly decrements or increments $R_{in}(c)$ by 1.

The transitions of the Generator do not affect the next property, since they do not cause any messages to be sent or received and all Clients created by the Generator are initially idle.

Therefore, all transitions in the system maintain the next property. □

THEOREM 5.26 (REACHABLE SYSTEM STATES)

The only reachable states for the SingleResourceMutualExclusion system are those where one of $\mathrm{P_{resource}}$, $\mathrm{P_{toclient}}$, $\mathrm{P_{client}}$ *or* $\mathrm{P_{toresource}}$ *holds.*

$$\textbf{invariant}.(P_{resource} \vee P_{toclient} \vee P_{client} \vee P_{toresource})$$

PROOF

Initially, the system is in a state where $P_{resource}$ holds, because all mailboxes are empty, no Client processes exist, and the Resource process is in the idle state. All possible sequences of state transitions from states where $P_{resource}$ hold establish only states where one of $P_{resource}$, $P_{toclient}$, P_{client} or $P_{toresource}$ holds, as demonstrated in Lemmas 5.22 through 5.25. Therefore, the only reachable states for the system are those where one of $P_{resource}$, $P_{toclient}$, P_{client} or $P_{toresource}$ hold. □

THEOREM 5.27 (GLOBAL SAFETY)

There is always exactly one live token in the SingleResourceMutualExclusion system.

$$\textbf{invariant}.(LT_{resource} + LT_{clients} + LT_{transit} = 1)$$

PROOF

We show that in every possible system state, the number of live tokens in the system is exactly 1.

In states where $P_{resource}$ holds, $LT_{resource} = 1$ because the Resource is in the idle state, $LT_{clients} = 0$ because *NOTBUSY* holds, and $LT_{transit} = 0$ because $R_{out} = 0$, $\langle \forall c \mid c \in CS \rhd -1 \leq R_{in}(c) + C_{out}(c) \leq 0 \rangle$, and $\langle \forall c \mid c \in CS \rhd C_{in}(c) = 0 \rangle$ hold. Therefore, the number of live tokens in the system is exactly 1 in states where $P_{resource}$ holds.

In states where $P_{toclient}$ holds, $LT_{resource} = 0$ because the Resource is in the busy state, $LT_{clients} = 0$ because *NOTBUSY* holds, and $LT_{transit} = 1$ because $R_{out} + \langle \Sigma c \mid c \in CS \rhd C_{in}(c) \rangle = 1$ and $\langle \forall c \mid c \in CS \rhd -1 \leq R_{in}(c) + C_{out}(c) \leq 0 \rangle$ hold. Therefore, the number of live tokens in the system is exactly 1 in states where $P_{toclient}$ holds.

In states where P_{client} holds, $LT_{resource} = 0$ because the Resource is in the busy state, $LT_{clients} = 1$ because $BUSY(R.current)$ holds, and $LT_{transit} = 0$ because $\langle \forall c \mid c \in CS \rhd -1 \leq R_{in}(c) + C_{out}(c) \leq 0 \rangle$, $R_{out} = 0$ and $\langle \forall c \mid c \in CS \rhd C_{in}(c) = 0 \rangle$ hold. Therefore, the number of live tokens in the system is exactly 1 in states where P_{client} holds.

In states where $P_{toresource}$ holds, $LT_{resource} = 0$ because the Resource is in the busy state, $LT_{clients} = 0$ because $NOTBUSY$ holds, and $LT_{transit} = 1$ because $\langle \forall c \mid c \in CS \wedge c \neq R.current \rhd$ $-1 \leq R_{in}(c) + C_{out}(c) \leq 0 \rangle$, $1 \leq R_{in}(R.current) + C_{out}(R.current) \leq 2$, $R_{out} = 0$ and $\langle \forall c \mid c \in CS \rhd C_{in}(c) = 0 \rangle$ hold. Therefore, the number of live tokens in the system is exactly 1 in stats where P_{client} holds.

Since we know from Theorem 5.26 that one of $P_{resource}$, $P_{toclient}$, P_{client} and $P_{toresource}$ holds in every possible system state, there is always exactly one live token in the SingleResourceMutualExclusion system. □

We have now shown that the SingleResourceMutualExclusion system fulfills our safety condition—at most one Client is using the Resource at any point during the system's execution. We now show that it fulfills our progress condition, by ensuring that all Clients that are in the waiting state eventually enter the busy state or leave the system.

LEMMA 5.28 (RESOURCE PROCESS SERVING PROGRESS)

The Resource in the SingleResourceMutualExclusion system always eventually becomes idle.

$$R.busy \leadsto R.idle$$

PROOF

Assume R is busy, and remains so forever. $R.current$ is a Client process, because only Client processes send messages to $R.requestIn$ (by inspection). We know from Theorem 5.3 that the difference between the number of releases received by R from $R.current$ and the number of tokens sent by R to $R.current$ is exactly the number of instances of $R.current$ in $R.releases$ minus 1. We know that the number of instances of $R.current$ in $R.releases$ is zero, because if it were not, R would become idle as shown in Theorem 5.6, contradicting our assumption. So the number of tokens sent by R to $R.current$ is exactly one greater than the number of releases received by R from $R.current$, and no future release is ever received by R from $R.current$ (since this would lead to R becoming idle).

Because of the messaging connections of Theorem 5.21, this means that the number of tokens received by $R.current$ must become exactly one greater than the number of releases sent by $R.current$ and remain that way forever. However, $R.current$ is a Client process, and Corollary 5.16 tells us that if a Client process has received more tokens than it has sent releases, it will eventually reach a state where it has received the same number of tokens as it has sent

releases. This is a contradiction, because it means that, eventually, R will receive another release from $R.current$. R therefore cannot remain busy forever. □

THEOREM 5.29 (GLOBAL PROGRESS)

All Client processes that are waiting to access the Resource in the SingleResourceMutualExclusion system eventually either get access to the Resource or leave the system.

$$\langle \forall p \mid p \in \mathcal{P} \wedge \langle \exists r \rhd p = \text{Client}(r) \rangle \rhd p.waiting \rightsquigarrow p.busy \vee p.gone \rangle$$

PROOF

Combining Theorems 5.5, 5.28 and 5.26 with the knowledge that request messages cannot be overtaken once they are delivered to the Resource's "requestIn" inbox tells us that every Client that sends a request (that is, every client that enters the waiting state) will eventually be sent a token. When this happens, the Client will have the opportunity to enter the busy state when that token reaches its inbox, fulfilling the leads-to condition. Additionally, transition (5) of the Client is an unfair transition that allows the Client to go from the waiting state to the gone state. This also fulfills the leads-to condition. Since the busy state will eventually result in all executions except those where the gone state results, the leads-to condition is always fulfilled.

□

Chapter 6

Nondeterministic Example 2: Dynamic Drinking Philosophers

Our third and final example Dynamic UNITY system is a solution to a particular resource allocation problem called the dynamic drinking philosophers problem. This example demonstrates many of Dynamic UNITY's features, as both clients and resources can enter and leave the system, and the set of resources required by a particular client can change. It is also far more complex than the previous example, which demonstrates the applicability of the Dynamic UNITY formalism to more complex problems.

6.1 Problem Statement

Many systems exist where a single client requires simultaneous exclusive access to multiple resources. For instance, a process running on a machine with a shared filesystem may need write access to several files, and may not be able to complete its computation until it gets this access. If this system is dynamic, both the set of running client processes and the set of files in the filesystem may change.

The drinking philosophers problem [7], or drinkers problem, is a generalization of the dining philosophers problem that models multiple resource mutual exclusion in static environments. For this problem, a system consists of a set of processes (the philosophers), each of which can be in one of three states: *tranquil*, *thirsty* and *drinking*. The only allowed state transitions are tranquil → thirsty → drinking → tranquil, and it is guaranteed that no philosopher remains in the drinking state forever. A nonempty set of resources (the beverages) is associated with each philosopher that is in the thirsty or drinking state, and this set remains unchanged until that philosopher becomes tranquil again; when a tranquil philosopher becomes thirsty, its set of beverages may change. The problem is to ensure that no two philosophers who have a beverage in common drink at the same time, and that every thirsty philosopher drinks eventually. This

is the same as ensuring that every process in a system eventually receives simultaneous access to all the resources it requires to complete its computation.

The dynamic drinking philosophers problem extends the drinkers problem to model dynamic environments in the following way: instead of a static set of philosophers and a static set of beverages, the system contains dynamic sets of both. Beverages may enter and leave the system, and so may philosophers. We allow a beverage to leave the system at any time when it is not being consumed, and we allow a philosopher to leave the system at any time. The progress condition in this system is different from that of the traditional drinkers problem: since beverages and philosophers can leave the system at any time, we guarantee that a thirsty philosopher will eventually drink or leave the system rather than guaranteeing that it will eventually drink. In addition, we don't guarantee that a thirsty philosopher receives all the beverages in its beverage set. Instead, we guarantee that the thirsty philosopher receives all the beverages in its beverage set that have not left the system. This is the same as ensuring that every process in a system eventually receives simultaneous access to all of its requested resources that still exist in the system. An alternative specification is one in which a process's request is cancelled when one or more of the resources it requests is no longer in the system; our choice of specification is predicated on the assumption that obtaining some required resources is often better than obtaining none.

We choose to solve this problem with an algorithm that uses tokens (which are also used in most solutions to the traditional drinkers problem) and monotonically increasing local clocks to handle mutual exclusion and request prioritization. These local clocks are not systemwide logical clocks in the sense of Lamport [33] or Jefferson [28]—systemwide logical clocks guarantee causality of message ordering, while our clocks only guarantee that requests occur with monotonically increasing timestamps.

Every beverage has a unique and indivisible token associated with it, and the holder of the token has exclusive access to that beverage. Every philosopher in the system is implemented as a separate Dynamic UNITY process with its own independent local clock, and local clock times are used to prioritize requests for beverages and prevent deadlock. When a philosopher becomes thirsty, it sends requests to all the beverages in its beverage set. When it receives the tokens corresponding to all its beverages, it has exclusive access to the beverage set and can drink.

The dynamic aspects of the system render the traditional drinking philosophers algorithm insufficient to ensure system progress, even with the addition of logical clocks. Clearly, if a philosopher were to simply disappear after requesting a set of beverages the system would deadlock, because the beverage tokens would eventually be sent to the vanished philosopher and would never be returned. Similarly, if a beverage were to leave the system while still

```
type
  Request: record {philosopher: process, timestamp: integer}
  Release: process
  Token: process
  Demand: process
  BLeave: process
  PLeave: process
  BeverageSetRequest: process
  BeverageSet: set {process}
```

Specification 6.1: Message types for the dynamic drinking philosophers system

being in the beverage sets of one or more philosophers the system would deadlock, because the philosophers would be waiting for tokens that would never arrive. To account for these situations in the dynamic system, we introduce an additional Dynamic UNITY process called the *coordinator* that coordinates the entry and exit of processes to and from the system. Much of the coordinator's functionality is analogous to that of a distributed directory service, and would be implemented as such in an actual distributed system; we choose to implement it as a single Dynamic UNITY process for simplicity.

Every message passed in the system contains either one token or no tokens; there are no multiple-token messages. In addition, every request message contains a timestamp that is used to establish the priority of requests from different philosophers. We assume that timestamps from different philosophers are unique (that is, that no two messages from different philosophers can have the same timestamp). This is a reasonable assumption, since it is always possible to break timestamp ties using a systemwide unique identifier as a tiebreaker.

We specify the components of our system individually and give a high level description of their behavior. We then prove a particular progress condition, namely that there exists a metric that ensures that a thirsty philosopher will eventually reach the drinking state. We do not carry out a full proof of the dynamic drinking philosophers system, as previous chapters have already demonstrated our proof method in detail.

We first specify some message types that will be used for communication among the processes in our system. This allows us to avoid replicating the same type-section in all three programs. For the purposes of this discussion, we assume all messages in the system are of one of these message types. This makes our programs less complicated, eliminating the need to filter out messages of other unexpected types that may arrive on the inboxes.

6.2 Message Types

The message types of Specification 6.1 allow processes in a dynamic drinking philosophers system to distinguish between messages that carry otherwise identically-typed information (6 of the 8 message types consist solely of a process reference). Request and Release are message types sent from a philosopher to a beverage, to request a token and release a token; Token and Demand are message types sent by a beverage to a philosopher, containing a token and a demand for the return of its token; BLeave is a message type sent by a beverage to the coordinator, and then by the coordinator to all philosophers in the system, announcing the beverage's departure from the system; PLeave is a message type sent by a philosopher to the coordinator and the beverages, informing them of its departure from the system; BeverageSetRequest is a message type sent by a philosopher to the coordinator requesting a new beverage set; and BeverageSet is a message type sent by the coordinator to a philosopher containing a new beverage set.

6.3 The Beverage Program

Each beverage (Specification 6.2) starts in a state where it is holding its own token. It continually receives requests from philosophers, and services them in the following way. The requests are sorted according to their timestamps, and the earliest timestamp is always serviced if possible. If a new request comes in with an earlier timestamp than the one that is currently being serviced, the beverage demands the token back from the philosopher to which it had last sent the token. In this way, a priority queue of philosophers is established. Since the philosophers always request their entire set of beverages with the same timestamp, this gives a global priority to each philosopher and prevents deadlocks that could be caused by priority cycles among philosophers competing for multiple beverages. A beverage can only leave the system when it holds its own token (that is, when it is guaranteed to not be in use by a philosopher), and when it does so, it sends a notification to the coordinator. This allows the coordinator to notify the philosophers, so they can remove the beverage from their beverage sets.

6.4 The Philosopher Program

Each philosopher (Specifications 6.3–6.5) starts in the tranquil state. On a transition to the thirsty state, it sends requests to all the beverages in its set and waits for the tokens corresponding to them. These requests are all stamped with the philosopher's current local time, called the *request time*, and this time is also recorded locally in the philosopher's state. It is pos-

program Beverage (coordinator: **process**)

declare
philosopherIn: **inbox**
requests: **set** {Request}
releases: **set** {Request}
current: Request
demandSent: **boolean**

always
idle $\triangleq$ current = $\perp$;
busy $\triangleq \neg$idle

initially
idle $\wedge$ requests = $\emptyset$ $\wedge$ releases = $\emptyset$ $\wedge$ demandSent = **false**

fair-transition
(1) philosopherIn.**probe** $\wedge$ philosopherIn.**current**.msg.**type** = Request $\longrightarrow$
requests$'$ = requests $\cup$ {philosopherIn.**current**.msg} $\wedge$
philosopherIn.**advance**
(2) [] philosopherIn.**probe** $\wedge$ philosopherIn.**current**.msg.**type** = Release $\longrightarrow$
releases$'$ = releases $\cup$ {philosopherIn.**current**.msg} $\wedge$
philosopherIn.**advance**
(3) [] idle $\wedge$ requests $\neq \emptyset$ $\longrightarrow$
$\langle \exists$ r | r $\in$ requests $\wedge$ r.timestamp = $\langle$min s | s $\in$ requests $\triangleright$ s.timestamp$\rangle$ $\triangleright$
current$'$ = r $\wedge$ requests$'$ = requests $\setminus$ {r} $\wedge$
send(r.philosopher, "beverageIn", Token(**this**))$\rangle$ $\wedge$
demandSent$'$ = **false**
(4) [] busy $\wedge$ $\langle \exists$ r | r $\in$ requests $\triangleright$ r.timestamp < current.timestamp$\rangle$ $\wedge$
$\neg$demandSent $\longrightarrow$
send(current.philosopher, "beverageIn", Demand(**this**)) $\wedge$
demandSent$'$ = **true**
(5) [] busy $\wedge$ $\langle \exists$ r | r $\in$ releases $\triangleright$ r.philosopher = current.philosopher$\rangle$ $\longrightarrow$
releases$'$ =
releases $\setminus$ {r | r $\in$ releases $\triangleright$ r.philosopher = current.philosopher} $\wedge$
current = $\perp$

unfair-transition
(6) idle $\longrightarrow$ **send**(coordinator, "in", BLeave(**this**)) $\wedge$ **stop**

end

Specification 6.2: The *Beverage* program, part of the dynamic drinking philosophers system

program Philosopher (coordinator: **process**, initialBeverages: **set** {**process**},
initialTime: **integer**)

declare
beverageIn, coordinatorIn: **inbox**
beverages: **set** {**process**}
requests: **set** {**process**}
tokens: **set** {Token}
demands: **set** {Demand}
requestTime: **integer**
clock: **integer**
tranquil, thirsty, drinking: **boolean**

always
gone $\triangleq$ $\neg$tranquil $\wedge\neg$thirsty $\wedge\neg$drinking

initially
beverages = initialBeverages $\wedge$ tokens = $\emptyset$ $\wedge$ demands = $\emptyset$ $\wedge$
clock = initialTime $\wedge$ tranquil = **true** $\wedge$ thirsty = **false** $\wedge$ drinking = **false**

fair-transition
(Specification 6.4)

unfair-transition
(Specification 6.5)

end

Specification 6.3: The *Philosopher* program, part of the dynamic drinking philosophers system

(1) beverageIn.**probe** ∧ beverageIn.**current**.msg.**type** = Token ⟶
    tokens′ = tokens ∪ {beverageIn.**current**.msg} ∧ beverageIn.**advance**
(2) [] beverageIn.**probe** ∧ beverageIn.**current**.msg.**type** = Demand ∧
    ⟨∃ t | t ∈ tokens ▷ t.beverage = beverageIn.**current**.msg.beverage⟩ ⟶
    demands′ = demands ∪ {beverageIn.**current**.msg} ∧ beverageIn.**advance**
(3) [] beverageIn.**probe** ∧ beverageIn.**current**.msg.**type** = Demand ∧
    ¬⟨∃ t | t ∈ tokens ▷ t.beverage = beverageIn.**current**.msg.beverage⟩ ⟶
    beverageIn.**advance**
(4) [] coordinatorIn.**probe** ∧ coordinatorIn.**current**.**type** = BLeave ⟶
    beverages′ = beverages \ {coordinatorIn.**current**.msg.beverage} ∧
    coordinatorIn.**advance**
(5) [] tranquil ∧ coordinatorIn.**probe** ∧ coordinatorIn.**current**.**type** = BeverageSet ⟶
    beverages′ = coordinatorIn.**current**.msg.beverages ∧ coordinatorIn.**advance**
(6) [] ¬tranquil ∧ coordinatorIn.**probe** ∧ coordinatorIn.**current**.**type** = BeverageSet ⟶
    coordinatorIn.**advance**
(7) [] thirsty ∧ ⟨∃ r | r ∈ beverages ▷ r ∉ requests⟩ ⟶
    requests′ = beverages ∧
    send(⟨, r | r ∈ beverages ∧ r ∉ requests ▷
                r, "philosopherIn", Request(**this**, requestTime)⟩)
(8) [] thirsty ∧ ⟨∀ r | r ∈ beverages ▷ ⟨∃ t | t ∈ tokens ▷ t.beverage = r⟩⟩ ⟶
    thirsty′ = **false** ∧ drinking′ = **true**
(9) [] drinking ⟶
    drinking′ = **false** ∧ tranquil′ = **true** ∧
    send(⟨, t | t ∈ tokens ▷ t.beverage, "philosopherIn", Release(**this**)⟩) ∧
    requests′ = ∅ ∧ demands′ = ∅ ∧ tokens′ = ∅
(10) [] ¬drinking ∧ ⟨∃ d, t | d ∈ demands ∧ t ∈ tokens ▷ t.beverage = d.beverage⟩ ⟶
    send(⟨, d | d ∈ demands ∧ ⟨∃ t | t ∈ tokens ▷ t.beverage = d.beverage⟩ ▷
                d.beverage, "philosopherIn", Release(**this**)⟩) ∧
    demands′ = demands \
        {d | d ∈ demands ∧ ⟨∃ t | t ∈ tokens ▷ t.beverage = d.beverage⟩} ∧
    tokens′ = tokens \
        {t | t ∈ tokens ∧ ⟨∃ d | d ∈ demands ▷ t.beverage = d.beverage⟩} ∧
    requests′ = requests \
        {r | r ∈ requests ∧ ⟨∃ d | d ∈ demands ▷ r.beverage = d.beverage⟩}
(11) [] clock′ = clock + 1

Specification 6.4: Fair transition section of the *Philosopher* program

(12) tranquil ⟶
    requests′ = beverages ∧ tranquil′ = **false** ∧ thirsty′ = **true** ∧
    requestTime′ = clock ∧ clock′ = clock + 1 ∧
    send(⟨, r | r ∈ beverages ▷ r, "philosopherIn", Request(**this**, clock)⟩)
(13) [] tranquil ∧ beverages ≠ ∅ ⟶
    beverages′ = ∅ ∧ **send**(coordinator, "in", BeverageSetRequest(**this**))
(14) [] tranquil ⟶ tranquil′ = **false** ∧ **send**(coordinator, "in", PLeave(**this**)) ∧ **stop**
(15) [] thirsty ∨ drinking ⟶
    thirsty′ = **false** ∧ drinking′ = **false** ∧
    requests′ = ∅ ∧ tokens′ = ∅ ∧ demands′ = ∅ ∧
    send(⟨, r | r ∈ requests ▷ r, "philosopherIn", Release(**this**)⟩) ∧
    send(coordinator, "in", PLeave(**this**)) ∧ **stop**

Specification 6.5: Unfair transition section of the *Philosopher* program

sible for the philosopher to receive a token and then subsequently have that token demanded back by the beverage associated with it; if the philosopher is not already drinking when this happens, it returns the token and makes another request with the same request time as it used originally. In this way, philosophers maintain their global priority relative to the other philosophers. When the philosopher holds the tokens of all its beverages, it enters the drinking state, and when it leaves the drinking state it sends the tokens back and enters the tranquil state. A philosopher can leave the system regardless of what state it is in, but it must send releases to any beverages in its beverage set for which it has outstanding requests (regardless of whether or not it holds the tokens for them). This prevents deadlock by ensuring that all the beverages will get their tokens back, or will not send them out in the first place if a release arrives before the corresponding request is serviced.

6.5 The Coordinator Program

---

program Coordinator

declare
in: **inbox**
beverages: **set** {**process**}
philosophers: **set** {**process**}
clock: **integer**

initially
beverages = ∅ ∧ philosophers = ∅ ∧ clock = 0

fair-transition
(1) in.**probe** ∧ in.**current**.msg.**type** = BLeave ⟶
beverages′ = beverages \ {in.**current**.msg.beverage} ∧
send(⟨, p | p ∈ philosophers ▷ p, "coordinatorIn", in.**current**.msg⟩)
(2) [] in.**probe** ∧ in.**current**.msg.**type** = PLeave ⟶
philosophers′ = philosophers \ {in.**current**.msg.philosopher}
(3) [] in.**probe** ∧ in.**current**.msg.**type** = BeverageSetRequest ⟶
s: s ⊆ beverages ∧
send(in.**current**.msg.philosopher, "coordinatorIn", BeverageSet(s))
(4) [] clock′ = clock + 1

unfair-transition
(5) beverages′ = beverages ∪ {**new** Beverage(**this**)}
(6) [] s: s ⊆ beverages ∧
philosophers′ = philosophers ∪ {**new** Philosopher(**this**, s, clock)} ∧
clock′ = clock + 1

Specification 6.6: The *Coordinator* program, part of the dynamic drinking philosophers system

The function of the coordinator (Specification 6.6) is to model the resource discovery and failure notification algorithms that would be used in an actual implementation of a dynamic drinking philosophers system. It is essentially a directory service—it creates and keeps track of all the system's beverages and philosophers, and is responsible both for assigning beverage sets to philosophers and for distributing notifications when a philosopher or beverage leaves the system. When a beverage leaves the system, all the philosophers in the system are notified of its departure so that the beverage can be removed from their beverage sets.

The coordinator, like the beverages and philosophers, has a local clock. This clock is used to initialize the local clocks of newly created beverages and philosophers and is important in preventing infinite overtaking of thirsty philosophers by new philosophers that enter the system.

6.6 The Composed System

---

system DynamicDrinkingPhilosophers

initial-program Coordinator

program Beverage (coordinator: **process**, initialTime: **integer**)

program Philosopher (coordinator: **process**, initialBeverages: **set** {**process**},
initialTime: **integer**)

end

Specification 6.7: The *DynamicDrinkingPhilosophers* system

The composed system (Specification 6.7) contains a single coordinator, instantiated as the initial process. We do not present a full proof of correctness for the system because most of the concepts that would be illustrated by such a proof have already been demonstrated in previous chapters. Instead, we give a partial proof that establishes the main progress property for the system.

6.6.1 Partial Proof of Progress

We present a partial proof of system progress by formulating and proving the validity of a progress metric that guarantees every thirsty philosopher will eventually drink or leave the system. This is only a partial proof because we use an assumption about the token passing behavior of the system in order to prove the correctness of our metric. In order to formulate our metric, we first define and prove properties of the quantities that will be used in the metric.

For the remainder of this section, we denote the set of philosophers in a drinking philosophers system by $\mathcal{P}$, and the coordinator process in the system by c.

We first define a quantity that reflects the difference between a philosopher's request time and the coordinator's local time, and prove that it is monotonically nonincreasing.

Definition 6.1 (Coordinator Difference) *The* coordinator difference *for a thirsty philosopher* p*, denoted* CD(p)*, is the difference between* p*'s request time and the coordinator's local time, or 0, whichever is greater.*

$$CD(p) \triangleq \max(p.requestTime - c.clock, 0)$$

Lemma 6.2 (Coordinator Difference Monotonicity)
CD(p) *is monotonically nonincreasing.*

$$\langle \forall k \rhd CD(p) = k \textbf{ next } CD(p) \leq k \rangle$$

Proof
We prove this next property by examining the effect of every transition of the system's programs on $CD(p)$, given that p is in the thirsty state.

Only two transitions of the Coordinator program, (4) and (6), modify $c.clock$. Both of these transitions increment the value of $c.clock$ by 1. Therefore, both of these transitions satisfy the next property by reducing $CD(p)$ (if it is greater than 0) or leaving it unchanged (if it is less than or equal to 0). The transitions that do not modify $c.clock$ satisfy the next property by leaving $CD(p)$ unchanged.

No transition of the Philosopher program modifies $p.requestTime$ when p is in the thirsty state. The one transition that modifies $p.requestTime$, (12), is enabled only in the tranquil state.

We have shown that every transition of the Coordinator and Philosopher programs either leaves $CD(p)$ unchanged or decreases it. No transition of any other program in the system can modify $CD(p)$, because it is calculated solely from state variables of the Philosopher and Coordinator programs. Therefore, the next property holds. □

Next, we define a quantity $PD(p)$ that reflects the maximum number of times a philosopher p can be overtaken by other philosophers currently in the system. We then show that the lexicographic pair $(CD(p), PD(p))$ is monotonically nondecreasing, and that its value must eventually decrease if it is greater than (0, 0).

Definition 6.3 (Philosopher Difference) *The maximum number of times a thirsty philosopher* p *can be overtaken by another philosopher* q *in the system is the difference between* p*'s request time and* q*'s local time, or 0, whichever is greater. The* philosopher difference *for a*

thirsty philosopher p*, denoted* PD(p)*, is the maximum number of times a philosopher* p *can be overtaken by all other philosophers in the system.*

$$PD(p) \triangleq \langle \Sigma q \mid q \in \mathcal{P} \rhd \max(p.requestTime - q.clock, 0) \rangle$$

LEMMA 6.4 ((CD, PD) MONOTONICITY)
The lexicographic pair (CD(p), PD(p)) *is monotonically nonincreasing.*

$$\langle \forall k, l \rhd (CD(p), PD(p)) = (k, l) \textbf{ next } (CD(p), PD(p)) \leq (k, l) \rangle$$

PROOF
We prove this next property by examining the effect of every transition of the system's programs on $(CD(p), PD(p))$, given that p is in the thirsty state. We have already shown that $CD(p)$ is monotonically nonincreasing. Therefore, all we must show is that if $PD(p)$ increases, there is a corresponding decrease in $CD(p)$.

We begin with the Philosopher program. Since no transition of p that is enabled when p is in the thirsty state changes the value of $p.requestTime$, execution of any transition of p leaves $PD(p)$ unchanged. For every philosopher q in the system, where $q \neq p$, the transitions have the following effects:

- Transitions (1) through (10), and (13), do not change $q.clock$ and do not remove q from the system. Therefore, they do not affect the value of $PD(p)$.
- Transitions (11) and (12) each increase $q.clock$ by 1, and do not remove q from the system. Therefore, they either decrease $PD(p)$ by 1 (if $p.requestTime > q.clock$) or leave its value unchanged (if $p.requestTime \leq q.clock$).
- Transitions (14) and (15) remove q from the system. Therefore, they either decrease $PD(p)$ (if $p.requestTime > q.clock$) or leave its value unchanged (if $p.requestTime \leq q.clock$).

We continue with the Coordinator program. The transitions of the Coordinator program have the following effects:

- Transitions (1) through (5) do not add new philosophers to the system. Therefore, they do not affect the value of $PD(p)$.
- Transition (6) increases $c.clock$ by 1 and adds a new philosopher to the system. The new philosopher's clock value is initialized to the value of $c.clock$. Therefore, if $p.requestTime > c.clock$, it increases $PD(p)$ by the difference between them and decreases $CD(p)$ by 1,

the net effect of which is to decrease $(CD(p), PD(p))$. If $p.requestTime \leq c.clock$, it leaves the values of both $CD(p)$ and $PD(p)$ unchanged.

We have shown that every transition of the Philosopher and Coordinator programs either leaves $(CD(p), PD(p))$ unchanged or decreases it. No transition of any other program in the system can modify $(CD(p), PD(p))$, because it is calculated solely from state variables of the Philosopher and Coordinator programs. Therefore, the next property holds. □

LEMMA 6.5 ((CD, PD) TRANSIENCE)
If $(\mathrm{CD(p)}, \mathrm{PD(p)}) > (0,0)$*, it will eventually decrease.*

$$\langle \forall k, l \mid k \geq 0 \wedge l \geq 0 \wedge (k,l) \neq (0,0) \triangleright \textbf{transient}.((CD(p), PD(p)) = (k,l)) \rangle$$

PROOF
We prove this transient property by examining the transitions of the Coordinator and Philosopher programs. Transition (4) of the Coordinator program is a fair transition that is always enabled, and that increases the value of $c.clock$ by 1. Transition (11) of the Philosopher program is a fair transition that is always enabled, and that increases the value of $q.clock$ by 1 for a philosopher q. Therefore, if $(CD(p), PD(p)) > (0,0)$, execution of one of these transitions will reduce its value. This is sufficient to prove the transient property. □

Finally, we define a quantity $HP(p)$ that reflects the number of thirsty philosophers in the system with higher priority than a particular philosopher p. We then show that the lexicographic triple $(CD(p), PD(p), HP(p))$ is monotonically nondecreasing.

DEFINITION 6.6 (HIGHER PRIORITY) *A thirsty philosopher* q *has a* higher priority *than another thirsty philosopher* p *if* q*'s request time is earlier than* p*'s request time. We denote the number of philosophers in the system that have higher priority than* p *by* HP(p)*.*

$$HP(p) \triangleq \langle \#q \mid q \in \mathcal{P} \wedge (q.thirsty \vee q.drinking) \triangleright q.requestTime < p.requestTime \rangle$$

LEMMA 6.7 ((CD, PD, HP) MONOTONICITY)
The lexicographic triple $(\mathrm{CD(p)}, \mathrm{PD(p)}, \mathrm{HP(p)})$ *is monotonically nonincreasing.*

$$\langle \forall k, l, m \triangleright (CD(p), PD(p), HP(p)) = (k,l,m) \textbf{ next } (CD(p), PD(p), HP(p)) \leq (k,l,m) \rangle$$

PROOF

We prove this next property by examining the effect of every transition of the system's programs on $(CD(p), PD(p), HP(p))$, given that p is in the thirsty state. We have already shown that $(CD(p), PD(p))$ is monotonically nonincreasing. Therefore, all we must show is that if $HP(p)$ increases, there is a corresponding decrease in $(CD(p), PD(p))$.

We begin with the Philosopher program. Since no transition of p that is enabled when p is in the thirsty state changes the value of $p.requestTime$, execution of any transition of p leaves $HP(p)$ unchanged. For every philosopher q in the system, where $q \neq p$, the transitions have the following effects:

- Transitions (1) through (7), (10), (11), and (13) do not change the state of q. Therefore, they do not affect the value of $HP(p)$.
- Transition (8) changes the state of q from thirsty to drinking. It does not affect the value of $HP(p)$, because $HP(p)$ counts both thirsty and drinking philosophers.
- Transition (9) changes the state of q from drinking to tranquil. It therefore either decreases $HP(p)$ by 1 (if $q.requestTime < p.requestTime$) or leaves its value unchanged (if $q.requestTime \geq p.requestTime$).
- Transition (12) changes the state of q from tranquil to drinking, and increments $q.clock$ by 1. Therefore, if $q.clock < p.requestTime$, it increases $HP(p)$ by 1 and decreases $PD(p)$ by 1, the net effect of which is to decrease $(CD(p), PD(p), HP(p))$. If $q.clock \geq p.requestTime$, it leaves the values of both $HP(p)$ and $PD(p)$ unchanged.

We have shown that every transition of the Philosopher program either leaves $(CD(p), PD(p), HP(p))$ unchanged or decreases it. No transition of any other program in the system can modify $HP(p)$, because it is calculated solely from state variables of the Philosopher program, and we have previously shown that $(CD(p), PD(p))$ is monotonically nondecreasing. Therefore, the next property holds. □

Having defined these quantities and proven monotonicity and transience properties for them, we can use them to establish a progress metric as follows.

THEOREM 6.8 (PROGRESS METRIC)

If p *is a thirsty philosopher in a dynamic drinking philosophers system, the triple* (CD(p), PD(p), HP(p)) *is a lexicographic progress metric for* p.

$$M(p) \triangleq (CD(p), PD(p), HP(p))$$

PROOF

To prove that $M(p)$ is a progress metric, we need to show that $M(p)$ has a lower bound, that $M(p)$ never increases, that if p doesn't drink and doesn't leave the system $M(p)$ eventually decreases, and that when $M(p)$ has reached its lower bound p is guaranteed to either drink or leave the system.

In this proof, we rely on properties of the Beverage and Philosopher programs that have not been explicitly proven elsewhere; in a complete proof of the system, we would need to prove these properties as well. In particular, we assume that if a philosopher retains the highest priority long enough, it eventually receives all its tokens (and therefore eventually drinks). This is a reasonable assumption, because we know that if a philosopher has the highest priority, there are no other philosophers in the thirsty or drinking states that have earlier request times. Therefore, that philosopher (which we call q) is either the current requestor or the head of the request queue for every beverage in its beverage set. If q is the current requestor for a beverage b, it either holds b's token or will hold it when it arrives. If q is the head of b's request queue and some other philosopher r is b's current requestor, b will demand the token back from r (if necessary; it may receive it before making the demand) and then send it to q, which will then be b's current requestor. Finally, if q is the head of b's request queue and b holds its token, q will become b's current requestor (if no other higher priority request arrives first). If no other philosopher takes the highest priority away from q, eventually q will be the current requestor for all its beverages and all the tokens for those beverages will reach q. If no demands are sent in the meantime, which must be the case if q remains the highest priority philosopher, this means that q will drink.

To show that $M(p)$ has a lower bound, we need only examine its definition. Each field of $M(p)$ clearly has a lower bound of 0: *CD* is the maximum of an integer value and 0, *PD* is a sum of such maxima, and *HP* is the number of processes in a system that satisfy certain constraints. Therefore, $M(p)$ has a lower bound at (0, 0, 0). This completes the first part of the proof.

From Lemma 6.7, we know that $M(p)$ never increases. This completes the second part of the proof.

Finally, we need to show that if p does not drink and does not leave the system, $M(p)$ eventually decreases, and that when $M(p)$ has decreased to (0, 0, 0), p is guaranteed to drink or leave the system.

From Lemma 6.5, we know that if $CD(p) > 0$ or $PD(p) > 0$, $M(p)$ eventually decreases (because $(CD(p), PD(p))$ eventually decreases). Therefore, if p does not drink or leave the system, $M(p)$ will eventually become (0, 0, k) for some $k \geq 0$.

When this happens, there are k philosophers in the system with higher priority than p (by definition), and each such philosopher q must have $M(q) = (0, 0, l)$, $l < k$. This is the

case because $q.requestTime < p.requestTime$ if q has higher priority than p, and therefore $CD(q) \leq CD(p)$ and $PD(q) \leq PD(p)$ by definition. Therefore, there must be one philosopher r in the system with $M(r) = (0,0,0)$. This philosopher is guaranteed to retain the highest priority until it drinks or leaves the system, because if another philosopher obtained higher priority, $M(r)$ would have to increase, and we have proven that it cannot do so.

We have already assumed that a philosopher that retains the highest priority long enough eventually receives all its tokens, so philosopher r must eventually drink or leave the system. When it does either, $HP(p)$ decreases. Therefore, if p does not drink and does not leave the system, $M(p)$ eventually decreases.

We have therefore shown both that $M(p)$ eventually decreases if p does not leave the system and that a philosopher p for which $M(p) = (0,0,0)$ is guaranteed to drink or leave the system. This completes the partial proof. □

Chapter 7

Implementation of Dynamic UNITY Systems

In this chapter, we present some basic techniques that can be used to transform Dynamic UNITY programs and systems into executable code. While our focus is on the Java programming language [30], the basic techniques presented are applicable to other programming languages as well.

We first describe the feasibility of implementing Dynamic UNITY systems. We then describe a framework of Java classes that provides the functionality of a Dynamic UNITY runtime system; a full implementation of this framework that runs on a single Java Virtual Machine is found in Appendix A. As an example of system translation using this framework, we present the translation of the simple Dynamic UNITY program discussed in Chapter 2. A larger example, the translation of the single resource mutual exclusion system from Chapter 5, is found in Appendix B.

7.1 Feasibility

Many systems that can be specified in Dynamic UNITY can also be successfully implemented on real computers, but there exist two classes of systems that cannot be implemented. One class is comprised of those systems, such as the prime number sieve described in Chapter 4, that would exceed any fixed amount of space if allowed to run long enough but that could otherwise be implemented. The other class is comprised of those systems that have either malformed or uncomputable transitions and can therefore not be implemented at all. Malformed transitions, described in Section 2.2.6, either have unsatisfiable postconditions or are quantified over an uncountable range. Uncomputable transitions, described in the same section, are transitions whose guard or postcondition is uncomputable. It is clear that we cannot implement either malformed or uncomputable transitions as executable code, no matter what programming lan-

guage or hardware platform we choose.

For systems in the former class, an implementation would (at least potentially) exhibit resource usage that grows without bound, and would therefore be of limited use. For systems in the latter class, we cannot create any implementation at all. Therefore, it is important to determine whether a Dynamic UNITY system belongs to one of these two classes before attempting an implementation.

This determination can be made by inspecting the program texts and correctness proofs of the Dynamic UNITY system in question. From the program texts we can, in most cases, immediately determine whether or not a system will contain an infinite set of processes running simultaneously, create infinitely large data structures, or require the computation of uncomputable predicates during its execution. From the correctness proofs we can determine whether all the transitions of the system have been proven satisfiable. Once we have made this determination, we may move on to actually translating the Dynamic UNITY system into a set of Java classes.

7.2 Translation

In order to successfully translate a Dynamic UNITY system into a set of Java classes, we need two important pieces of infrastructure. The first is a way of instantiating new processes. For implementation on a single Java Virtual Machine, we can use a Java thread to represent each process, but for implementation on a network of Java Virtual Machines we need an underlying layer that handles process instantiation over a network. The Caltech Infospheres Infrastructure [5, 6], a preliminary to this work, is an example of such an underlying layer where the network in question is the Internet. Similar layers can be built using core Java technologies such as RMI and Object Activation [67] or any of a number of third-party middleware products. We will use syntax similar to Dynamic UNITY's syntax for process creation, with the justification that we can always implement a Java wrapper class that provides Dynamic UNITY process creation syntax over any underlying process creation infrastructure.

The second required piece of infrastructure is a message passing system that provides semantics equivalent to those of the Dynamic UNITY message passing system. One such system is `info.net`, the messaging layer from the Infospheres Infrastructure; another, ÜberNet [68], was implemented as a preliminary to this work. Both of these message passing systems implement first-in first-out message sequencing between every outbox/inbox pair (in the case of ÜberNet, other message sequencing semantics are also available). Since the Dynamic UNITY message passing system does not include the concept of multiple outboxes per process, we will explicitly use a single outbox for each process in our Java translation. We will otherwise use

syntax similar to Dynamic UNITY's syntax for messaging (assuming again that we can provide appropriate wrapper classes over any underlying infrastructure).

Once we have the infrastructure necessary for process instantiation and message passing, we must translate the Dynamic UNITY data types into Java types and classes. Most Dynamic UNITY data types—all except **array**, **inbox**, **process**, **record** and **multiset**—have direct analogues in the standard Java distribution. The Java classes *Serializable*, *Boolean*, *BigInteger*, *BigDecimal*, *String*, *List* and *Set* are exactly analogous to Dynamic UNITY types **any**, **boolean**, **integer**, **real**, **string**, **sequence** and **set**. We assume the existence of additional classes *Inbox* (part of our wrapper over the messaging infrastructure), *Process* (part of our wrapper over the process creation infrastructure), and *Multiset* that are analogous to the Dynamic UNITY types **inbox**, **process** and **multiset** respectively. We implement each instance of the Dynamic UNITY **record** type as a Java class containing data members corresponding to those in the record instance, and each instance of the Dynamic UNITY $\textbf{array}^{dim}$ type as a nested *List* with dimension *dim*—that is, an $\textbf{array}^2$ would be a *List* in which every element is a *List*.

Throughout this chapter, we make the assumption that the Java classes comprising Dynamic UNITY programs and systems are appropriately packaged; that is, our classes are defined in Java packages such that their names do not conflict with the names of already-existing Java classes. We also ignore exception handling, scoping keywords such as `private` and `protected`, and method synchronization primitives. In Appendix A, we provide full, compilable source code for a complete set of Dynamic UNITY runtime classes. That set of classes implements the interfaces we describe in the remainder of this chapter, including all exception handling and other constructs required by the Java language and runtime.

7.2.1 Translation of Transitions

The translation of Dynamic UNITY transitions into Java is straightforward in most cases, because a typical transition will translate to either an assignment statement or a set of assignment statements. In cases where nondeterminism is present (disjunction in the postcondition), any implementation—or a random choice among multiple implementations—that establishes the postcondition is acceptable. The Dynamic UNITY operations **stop** and **send** are implemented with the `stop()` and `send()` methods of *Program*, the base class for Dynamic UNITY programs discussed in the next section. The creation of new processes is implemented using a static method of the *Process* class called `instantiate()`, which takes as parameters a class name (analogous to a program name) and an array of *Serializable* objects to be passed as parameters to the new process.

EXAMPLE 7.1 (JAVA TRANSLATIONS OF DYNAMIC UNITY TRANSITIONS)

The following are some typical Dynamic UNITY transitions, along with one or more translations of each into Java. Note that there may be more valid translations for each than are listed here.

1. A variable increment:

   true $\longrightarrow x' = x + 1$

   ```
   x = x.add(BigInteger.ONE);
   ```

2. A conditional variable increment:

   $y > 5 \longrightarrow x' = x + 1$

   ```
   if (y.compareTo(BigInteger.valueOf(5)) > 0)
   {
     x = x.add(BigInteger.ONE);
   }
   ```

3. An intertwined variable increment (x and y are integer variables):

   true $\longrightarrow x' = y + 5 \wedge y' = x + 5$

   ```
   BigInteger temp = x.add(BigInteger.valueOf(5));
   x = y.add(BigInteger.valueOf(5));
   y = temp;
   ```

4. A nondeterministic variable increment (x is an integer variable):

   true $\longrightarrow x' > x + 5$

   ```
   x = x.add(BigInteger.valueOf(6));
   ```

   ```
   x = x.add(Math.abs(random.nextInt()) + 5);
   ```

5. A message send (dest is a variable of type process):

   true $\longrightarrow$ send(dest, "Vorlon", "Kosh")

   ```
   send(dest, "Vorlon", "Kosh");
   ```

6. A guarded message receive (box is an inbox, msg is a variable of type any):

   box.**probe** $\longrightarrow$ msg$'$ = box.**current**.msg $\wedge$ box.**advance**

```
if (box.probe())
{
  msg = box.getCurrent().getMessage();
  box.advance();
}
```

7. A process instantiation (the new process takes 2 parameters of type any, and a and b are variables of some type):

   true $\longrightarrow$ p: p′ = new ExampleProcess(a, b)

   ```
   Serializable[] parameters = new Serializable[2];
   parameters[0] = a;
   parameters[1] = b;
   Process.instantiate("ExampleProcess", parameters);

   Process.instantiate("ExampleProcess", new Serializable[2] {a, b});
   ```

8. A quantified transition (s is a set of integers, its Java equivalent is a sorted set, and `last` is the most recent element of the set selected for execution; the method `tailSet(i)` on a sorted set `s` returns the subset of `s` containing all members of `s` that are greater than `i`):

   $\langle [] \; i \mid i \in s \rhd s' = s \setminus \{i\} \rangle$

   ```
   if (s.isEmpty())
   {
     return;
   }
   SortedSet tailSet = s.tailSet(last.plus(BigInteger.ONE));
   if (tailSet.isEmpty())
   {
     last = (BigInteger) s.first();
   }
   else
   {
     last = (BigInteger) tailSet.first();
   }
   s.remove(last);
   ```

7.2.2 Translation of Programs

To simplify the process of translating Dynamic UNITY programs, we use a Java base class, *Program*, which includes functionality common to all Dynamic UNITY programs: implementations

of **stop** and **send** operations and execution of transitions from the fair- and unfair-transitions sections. All translations of Dynamic UNITY programs will be subclasses of *Program*. The interface for the *Program* class is as follows (all methods in this interface are called only by an instantiation of a *Program* subclass on itself):

- `void send(Process process, String inbox, Serializable message)` enqueues a copy of `message` for transmission to the destination inbox specified by `process` and `inbox`. This implements the **send** operation as applied to a single message. Multiple message sends are performed by calling this method multiple times.
- `void stop()` stops the process's execution. This implements the **stop** operation.
- `Process getProcess()` returns the process's reference. This implements the **this** operation.
- `void initialize()` can be overridden by subclasses to perform initialization tasks (implementing the initially-section of the program), if necessary.
- `void fairTransition()` must be overridden by subclasses to implement the fair transition set. The implementation should pick one fair transition, in a manner consistent with weak fairness, and execute it.
- `void unfairTransition()` can be overridden by subclasses to implement the unfair transition set, if necessary. The implementation should pick one unfair transition and execute it.
- `boolean unfairCondition()` can be overridden by subclasses to implement a scheduling policy for unfair transitions. It is called every time the process is selected for execution; if it returns **true**, an unfair transition is executed, and otherwise a fair transition is executed. In order to satisfy Dynamic UNITY execution semantics, `unfairCondition()` must be constructed such that it will return **false** an infinite number of times when it is called infinitely often. The default implementation always returns **false** so that no unfair transitions are ever executed.

In order to translate a particular Dynamic UNITY program into Java, the *Program* class must be subclassed. This subclass must conform to the following restrictions:

1. It has a single constructor, which takes one or two parameters. The first parameter is always a *Process* object and the second parameter, present only for Dynamic UNITY programs that take parameters, is an array of *Serializable* objects. This constructor must

call the superclass constructor with the *Process* object. The call to the superclass constructor allows the *Program* superclass to initialize itself properly by creating its *Outbox* and keeping a record of its associated *Process* object. If it takes an array of *Serializable* objects containing the parameters to the Dynamic UNITY program as a second parameter, the constructor must store these objects in some fashion so that they are accessible to the program when it begins execution.

2. If the Dynamic UNITY program being implemented has an initially-section, it must implement the `initialize()` method in such a way that it initializes the Dynamic UNITY state variables appropriately.

3. Its `fairTransition()` method is implemented such that all the fair transitions of the Dynamic UNITY program are chosen in a manner consistent with weak fairness. One way of doing this is to number the transitions and execute them in a round-robin fashion, so that the first time the method is called it executes transition 0, the next time it executes transition 1, etc. Another way of scheduling the transitions is to use a pseudorandom number generator which is known to be periodic and to cover its range, and determine the sequence of selected transitions from the generated random number sequence. Quantified transitions should generally be treated as a single transition at the fair transition level; when a quantified transition is selected, its scheduling can be handled either by selecting all the transitions in the quantification in sequence or by selecting one in a weakly fair manner.

4. Its `unfairTransition()` and `unfairCondition()` methods are implemented such that the unfair transitions of the Dynamic UNITY program are chosen in arbitrary fashion. One reasonable translation for any Dynamic UNITY program is the translation which leaves out the unfair transitions entirely, since executions of Dynamic UNITY programs never have to select unfair transitions. Another reasonable translation is to turn the unfair transitions into fair transitions and subject them to the same weak fairness constraint as the fair transition set. For robustness of possible execution sequences, these methods can be implemented such that unfair transitions are chosen according to various criteria. Example implementations include making `unfairCondition()` return **true** only when the current time in milliseconds from the epoch is divisible by 100000, or using a random number generator to determine whether an unfair transition gets executed at each step.

To illustrate the translation of a complete Dynamic UNITY program, we translate the program presented in Example 2.12 into Java:

Example 7.2 (Translation of a single Dynamic UNITY program to Java)
The original Dynamic UNITY program is as follows:

program ExampleSystemComponent
  declare
    theSet: **set** {integer}
  initially
    theSet = ∅
  fair-transition
    (1) **true** ⟶ ⟨∃ i | i ∉ theSet ▷ theSet′ = theSet ∪ {i}⟩
    (2) [] **true** ⟶ **stop**
    (3) [] ⟨[]i | i ∈ theSet ▷ i ≥ 731 ⟶ p: p′ = **new** ExampleSystemComponent⟩
end

The program translated to Java is seen below. We assume that the necessary packages for the Java classes we use are properly imported, and that the Random class implements a pseudorandom number generator which is periodic and covers its range.

Class 7.1 (A translation of an example program)

```
public class ExampleSystemComponent extends Program
{
  // Instance Variables

  int transition;
  BigInteger last;
  Random random;


  // Dynamic UNITY Variables

  Set theSet;


  // Constructor

  public ExampleSystemComponent(Process process)
  {
    super(process);

    theSet = new TreeSet();

    last = BigInteger.ZERO;
    random = new Random();
    start();
  }
```

```
  // Instance Methods

  public void fairTransition()
  {
    transition = Random.nextInt(3);

    switch (transition)
    {
      case 0:
      {
        BigInteger temp = new BigInteger(random.nextInt(), random);

        while (theSet.contains(temp))
        {
          temp = new BigInteger(random.nextInt(), random);
        }

        theSet.add(temp);
        break;
      }

      case 1:
      {
        stop();
        break;
      }

      case 2:
      {
        if (theSet.isEmpty())
        {
          break;
        }

        SortedSet tailSet = theSet.tailSet(last.plus(BigInteger.ONE));

        if (tailSet.isEmpty())
        {
          last = (BigInteger) theSet.first();
        }
        else
        {
          last = (BigInteger) tailSet.first();
        }

        if (last.compareTo(BigInteger.valueOf(731)) > 0)
        {
          Process newProcess =
            Process.instantiate("ExampleSystemComponent");
        }
      }
    }
  }
}
```

We implement the weak fairness in our fair transition section by choosing the transitions according to the sequence generated by our random number generator. In our handling of the quantified transition, we use a Java class that implements a sorted set (*TreeSet*) and keep track of the most recent value in the set whose corresponding transition was executed. This allows us to iterate through the set in a fair manner, returning to the lowest value in the set after executing the transition corresponding to the highest. This technique works well for quantified transitions in general, provided that they are quantified over data types for which an ordering relation exists.

7.2.3 Translation of Systems

To simplify the process of translating Dynamic UNITY systems, we use a Java base class, *System*, which includes functionality common to all Dynamic UNITY systems. Since a Dynamic UNITY system is comprised of a set of a programs and the labelling of one of those programs as "initial," the *System* class is very small: all it contains is a constructor which instantiates the initial program of a Dynamic UNITY system (using the *Process* class). All translations of Dynamic UNITY systems will be subclasses of *System* that, in their constructors, call the superclass constructor with the name of the initial program and the initial parameter list (if any). A Dynamic UNITY system is then started by simply instantiating an object of the appropriate class. Such subclasses will usually also contain a `main()` method, allowing them to be started directly from a command-line environment.

To illustrate the translation of a Dynamic UNITY system, we translate the example system from Example 2.12, which consists only of the initial program ExampleSystemComponent, into Java.

Example 7.3 (Translation of a Dynamic UNITY system to Java)
The original Dynamic UNITY system (omitting the definition of the component program) is as follows:

system ExampleSystem
  initial-program ExampleSystemComponent
end

The system translated to Java is as follows:

Class 7.2 (A translation of an example system)

```
public class ExampleSystem extends System
{
  // Constructor

  public ExampleSystem()
  {
    super(‘‘ExampleSystemComponent’’, null);
  }
}
```

More complex implementations of Dynamic UNITY systems are possible. For example, it might be reasonable to construct an implementation of a system which keeps track of all the processes running within it. However, more complex implementations are not necessary for correctness.

Chapter 8

Related Work

In this chapter, we outline some related work and contrast Dynamic UNITY with some other specification and proof methods for distributed systems.

8.1 Specification Methods

There are many different approaches to the specification of computer programs, both sequential and concurrent. Some of these form the basis of our approach to the specification of dynamic distributed systems.

8.1.1 Axiomatic Specification

An axiomatic specification defines fundamental language constructs by axioms, which are then used with inference rules to build more complicated language constructs. This approach was first used by Floyd [19], and has been applied to sequential systems with great success. The most commonly used forms of axiomatic specification for sequential programs are Hoare triples [25, 26] and Dijkstra's weakest preconditions [16, 17].

Axiomatic specification has also been applied to concurrent systems. For instance, Martin [42] gave an axiomatic definition of synchronization primitives in terms of boundedness, progress, and fairness, Owicki and Gries [54, 53] extended Hoare triples with the requirement to establish noninterference between threads of execution, and Lamport [34] extended Dijkstra's weakest preconditions with the notion of weakest and strongest invariants. Our assertions (described in Section 3.1.2) are conceptually similar to Hoare triples.

Another method of specifications for concurrent systems is to define the behavior of a particular component of the system given that the component's environment (the rest of the system) behaves in a specific way. That is, a particular component is required to behave correctly only if all other components in the system also do so. Various methods for specifying

component behavior in this way have been proposed, including rely-guarantee [29], hypothesis-conclusion [8], assumption-commitment [14], offers-using [32], and assumption-guarantee [1]. Each of these methods imposes different restrictions on the types of behavior which can be specified; for instance, Abadi and Lamport's assumption-guarantee approach restricts the assumptions on environment behavior to safety properties only. Chandy and Sanders's "weakest guarantee" method [9, 10] considers requirements on the entire system rather than just on the environment of a particular component. It divides component properties into two types: "exists-component," properties that hold for the entire system if they hold for a single component, and "all-component," properties that hold for the entire system only if they hold for all components. The properties we prove about Dynamic UNITY programs "in isolation" (that is, in arbitrary environments) are examples of "exists-component" properties.

8.1.2 Temporal Logic

Temporal logic [60, 62] is a branch of modal logic which contains temporal operators in addition to the standard propositional logic operators ($\wedge$, $\vee$, $\neg$, $\Rightarrow$). The use of temporal logic for reasoning about computer system executions was first proposed by Kröger [31] for sequential systems, and by Pneuli [58] for concurrent systems. Computations are viewed as sequences of global states, and properties of the systems to which these sequences correspond are specified as temporal properties of the sequences. Different versions of temporal logic arise from different formulations of the sets of sequences to which the temporal operators apply. For example, sequences may be linear or branching, and may be finite or infinite. One commonly-used version, sometimes called Manna-Pneuli theory [41, 40], is based on linear temporal logic.

There are multiple specification methods that incorporate temporal logic, including UNITY, about which we have already written in detail. Lamport's Temporal Logic of Actions (TLA) [35] is a specification logic based on the fundamental temporal logic operators $\Box$ ("always") and $\Diamond$ ("eventually"). TLA allows for the specification of both weak and strong fairness requirements (as well as allowing specifications without fairness requirements), and also allows for stuttering steps (as in UNITY and Dynamic UNITY). TLA+ [36] is a language for the specification of concurrent systems that uses TLA as its logical foundation.

The fundamental operators of Dynamic UNITY (**initially**, **next** , **transient**) are based on well-known operators in temporal logic. The derived operators of Dynamic UNITY (**stable**, **invariant**, **leads-to**) can also be found in many temporal logic-based formalisms, while the **follows** operator was originally presented by Sivilotti [66].

8.1.2.1 UNITY Variants

The UNITY language and logic has been extended and modified for various purposes since its publication. One notable such extension is Mobile UNITY [63, 57], which aims to use UNITY-based reasoning for systems with mobile processes. Mobile UNITY includes the concept, which we use in Dynamic UNITY, of systems with component programs. It also includes special operators for specifying component locations and the interactions between components which share the same location.

Misra has modified UNITY since its original publication, most notably replacing the safety operator **unless** with **co** (which is equivalent to Dynamic UNITY's **next**) [47]. Misra has also introduced a new multiprogramming model, called Seuss [48], which uses UNITY as its logical foundation.

8.1.3 Other Specification Methods

The actor model of computation, originally proposed by Hewitt [24] and studied extensively by Agha [2], shares many characteristics with Dynamic UNITY's execution model. An actor is a computational agent which communicates with other actors via fair asynchronous message passing: every actor has a single "mail address" and a set of actions that can be executed in response to receiving messages directed toward its mail address. These actions can include the sending of messages to other actors whose addresses are known and the creation of new actors. In addition, when receiving a message an actor must always specify a "replacement"—an actor that will accept the next incoming message—that can process the next incoming message even while the current one is still being processed by the original actor. Concurrency in the actor model arises from the sending of multiple messages in response to a single incoming message, and from the simultaneous handling of messages by actors and their replacements.

There are several key differences between actor computations and Dynamic UNITY executions: communication in Dynamic UNITY is more flexible, allowing multiple inboxes per process instead of a single mail address; execution in Dynamic UNITY consists of atomic transitions, which makes it unnecessary to explicitly handle race conditions and similar concurrency issues; execution of transitions in Dynamic UNITY occurs continually according to a weak fairness requirement, rather than solely in response to incoming messages; and Dynamic UNITY processes can remove themselves from a running system during execution.

Milner's Calculus for Communicating Systems (CCS) [43] is an algebraic process calculus for the specification of systems of communicating processes. The π-calculus [44, 45] is a generalization of CCS that allows the description of processes whose communication links change during execution, which makes it well suited to the description of systems with mobile

components.

I/O automata [39, 38] model interacting distributed system components by simple state machines in which the transitions are associated with named actions. These actions can be input actions, output actions, or internal actions. The first two are used for communication with an automaton's environment, while the last is used for state changes within the automaton itself. I/O automata are similar to Dynamic UNITY programs, in that they contain local state and transitions. However, the I/O automata model does not allow for the dynamic instantiation and destruction of automata, nor for changes in the communication patterns of already-existing automata analogous to inbox creation.

8.2 Communication Models

Various communication models have been used to design and reason about systems of communicating processes. Hoare's Communicating Sequential Processes (CSP) [26, 27] is a widely used model in which processes communicate via synchronous channels (with one channel connecting each pair of processes). A process sending a message to another process blocks until the other process executes a receive action, and vice versa. Misra and Chandy [49] used a variant of this model (with communications addressed to channels rather than to processes) to present a proof method for networks of processes.

Generative communication [21] is the model of communication that underlies the Linda [20] distributed programming language. In this model, messages are added as named tuples to a shared "tuple space" where they remain until a process receives them. Among the applications that have been implemented in Linda since its introduction is LIME [56], a system for developing mobile applications whose formal semantic definition has been verified using Mobile UNITY.

The **follows** operator, introduced by Sivilotti [66], is a natural choice for characterizing the behavior of asynchronous message passing systems upon which we have relied heavily in the definition of the Dynamic UNITY message passing system. A similar operator, the *observation*, is presented by Charpentier [12, 13].

Brock and Ackerman [4] present a proof that history relations (relations from input sequences to output sequences for message-passing processes) are insufficient to characterize the behavior of non-determinate message passing processes; they show that temporal relationships between input messages and output messages must also be considered when characterizing process behavior. Our proofs use such temporal information, primarily in the form of **follows** properties, to characterize the behavior of Dynamic UNITY processes.

8.3 "Stop" as a Failure Model

Various models have been proposed to reason about the behavior of computer systems which exhibit failures. Though we have not explicitly addressed failure models, the **stop** command in the Dynamic UNITY language is similar to a particular failure model, Schlicting and Schneider's [64] "fail-stop processor." A fail-stop processor automatically halts execution in the presence of a failure, before that failure becomes visible to external observers, and allows execution to be restarted on a working processor. The "volatile" storage of a fail-stop processor is lost during a failure, while the "stable" storage is unaffected. In contrast, the volatile state of a Dynamic UNITY process can be changed by the message-passing system after a **stop** occurs, while its non-volatile state remains unchanged. There is no way to restart a stopped process in the current Dynamic UNITY language, so Dynamic UNITY's **stop** does not completely model a fail-stop processor. However, it can be used as a starting point for defining such a model.

Chapter 9

Conclusion

9.1 Summary

In this thesis we have described Dynamic UNITY, a new specification language and proof logic for dynamic distributed systems. Our language and logic are extensions to the established UNITY formalism, which enables us to apply proof techniques developed for UNITY in our correctness proofs. In creating Dynamic UNITY, we made the following changes to the UNITY language and proof logic:

- We introduced modularity, by making programs components of larger systems rather than complete system specifications. This allows programs to be used as parts of various distinct systems, without any changes to their specifications or proofs of correctness.
- We added the ability for processes to create new processes, and to destroy themselves, at execution time. This enables us to design dynamic systems, which adapt to changing conditions or requirements during their execution, in an intuitive fashion.
- We eliminated shared variables entirely, providing a measure of information hiding for programs. This makes it possible to prove both safety and progress properties of a program that hold for every possible instantiation of that program regardless of the behavior of external processes. This facilitates the construction of modular correctness proofs for complex systems.
- We introduced a reliable asynchronous message passing layer, as an integral part of the new execution model. This eliminates the need for system architects to simulate message channels with history variables, queues, or other mechanisms, and therefore simplifies the construction of complex systems.
- We changed the notation for state transitions from guarded parallel assignment statements to guarded binary predicates. This adds some flexibility to the language, and al-

lows us to focus on the desired results of transition statements rather than on the precise set of assignment statements required to bring about those results. It also allows us to more easily and intuitively incorporate nondeterministic behavior into our designs.

We have demonstrated the utility of Dynamic UNITY by designing and proving the correctness of two example systems based on well-known algorithms—an infinite prime number sieve and a single resource mutual exclusion system. As an example of a more complex distributed systems problem, we presented a new, dynamic variant of the drinking philosophers problem; we then designed and partially proved an example system that solves this problem. All three of these example systems are dynamic, with component processes that enter and leave the system at runtime.

We have also presented a method for determining whether a given Dynamic UNITY specification can be transformed into an actual implementation in a standard programming language, and a method for transforming those specifications that can. This makes Dynamic UNITY a practical language for programming dynamic distributed systems, since a correct implementation can be guaranteed for any implementable Dynamic UNITY system.

9.2 Future Directions

There are many possible avenues of research to be pursued as extensions to this work. All of them build on the formalism we have defined, and further our goal of enabling the construction of correct dynamic distributed systems.

First, we could design and implement a compiler capable of transforming implementable Dynamic UNITY specifications into either straight executable code or code in another high-level language (such as Java). This would allow system designers to prototype in Dynamic UNITY directly, and then later to perform optimizations on the implementations generated by the compiler. It would also ensure the correctness of the generated implementations by eliminating the possibility of human error in code transformation.

Another possibility is to implement tools to assist in formulating correctness proofs of Dynamic UNITY systems. Theorem proving tools and environments, such as HOL [22], Coq [61], Isabelle [55] and Nuprl [15], might be useful as a basis for such implementation. In fact, some work on proving the correctness of UNITY systems using these tools has already been done (such as HOL-UNITY [3]). Such tools would be very helpful, both by assisting in proof formulation and in checking the validity of previously existing proofs.

There is also the possibility of using Dynamic UNITY to carry out proofs for programs in other high-level programming languages (such as C++ or Java). We have already demonstrated

the translation of Dynamic UNITY programs to code in high-level languages, but translation in the other direction is potentially even more useful. However, significant difficulties may arise in performing this translation, such as dealing with complex control structures and exception handling mechanisms.

Other possibilities arise when we consider potential future modifications to Dynamic UNITY itself. These can be roughly divided into two categories: modifications which aid the construction and proof of large systems, and modifications which add capabilities to the Dynamic UNITY language itself.

In the first category, we could introduce the ability to construct systems hierarchically, by building "subsystems" comprised of a set of programs and a set of exported mailboxes. Just as with programs, we could prove safety and progress properties about these subsystems in arbitrary environments, and then integrate them into larger subsystems or top-level systems. We could also devise a method for refinement, which would allow us to prove that particular Dynamic UNITY specifications are refinements of other, higher-level specifications (and vice-versa).

In the second category, we could introduce the object-oriented notion of inheritance, giving the ability for programmers to extend existing Dynamic UNITY programs while preserving some or all of their safety and progress properties. We could also introduce a stronger type system, including typing of inboxes, which would allow programmers to worry less about receiving messages of unexpected types from the environment and therefore reduce the work involved in proving the behavior of programs in arbitrary environments.

Appendix A

Java Implementation of a Dynamic UNITY Runtime Framework

The following Java classes comprise a complete Dynamic UNITY runtime framework that runs in a single Java Virtual Machine. It implements the interface described in Chapter 7, and can be used to build the example programs in that chapter and in Appendix B.

This runtime implementation does not use true asynchronous messaging; instead, it simulates asynchronous messaging by setting explicit arrival times for messages delivered to inboxes. This is done primarily for efficiency; since all the messages are staying within the same Java Virtual Machine, it is more efficient to simulate asynchronous delivery than to actually use additional Java threads to implement true asynchronous delivery.

The following sections each contain one of the seven classes that comprise the runtime system, with a brief description of the class functionality. The Javadoc comments for these classes have, for the most part, been left in the source code. This provides documentation for the actual implementation choices that have been made.

A.1 System

System is the base class for translations of Dynamic UNITY systems. It implements the functionality described in Section 7.2.3.

CLASS A.1 (*System*, PART OF A DYNAMIC UNITY RUNTIME FRAMEWORK)

```
package dynamicunity;

import java.io.Serializable;

public class System
{
  // Constructors
```

```
  public System(String initialProgram, Serializable[] initialParameters)
  {
    Process.instantiate(initialProgram, initialParameters);
  }


  public System(String initialProgram)
  {
    this(initialProgram, null);
  }
}
```

A.2 Process

Process is the class that encapsulates a Dynamic UNITY process identifier. It also includes utility methods for the creation and destruction of processes.

CLASS A.2 (*Process*, PART OF A DYNAMIC UNITY RUNTIME FRAMEWORK)

```
package dynamicunity;

import java.io.Serializable;
import java.lang.reflect.Constructor;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

public class Process implements Serializable
{
  // Static Variables

  /**
   * An empty array of Serializable, used when constructing new processes.
   **/

  public static final Serializable[] EMPTY_SERIALIZABLE_ARRAY =
    new Serializable[0];


  /**
   * A map from the system's process references to threads.
   **/

  private static Map processTable =
    Collections.synchronizedMap(new HashMap());


  /**
   * A map from the system's threads to process references.
   **/
```

```
private static Map threadTable =
  Collections.synchronizedMap(new HashMap());


/**
 * An integer value used to create unique process names.
 **/

private static int processIDUniquenessValue = 0;


// Instance Variables

/**
 * The ID string for this Process.
 **/

private String id;


// Static Methods

/**
 * Creates a new Dynamic UNITY process from the specified Dynamic UNITY
 * program with the specified instantiation parameters, and returns a
 * Process object identifying it.
 *
 * @param programName The fully qualified Java class name of the class
 *  that implements the Dynamic UNITY program to be instantiated.
 * @param parameters An array of Serializable values to be used as
 *  instantiation parameters.
 *
 * @return a Process object identifying the new process, or null if
 *  something went wrong during the instantiation of the new process.
 *
 * @concurrency (GUARDED)
 **/

public synchronized static Process
  instantiate(String programName, Serializable[] parameters)
{
  try
  {
    Class programClass = Class.forName(programName);
    Constructor[] constructors = programClass.getConstructors();
    Program newProgram = null;
    Process newProcessID = getUniqueProcessID();

    Object[] realParameters;

    if (parameters == null)
    {
      realParameters = new Object[1];
```

```
      realParameters[0] = newProcessID;
    }
    else
    {
      realParameters = new Object[2];
      realParameters[0] = newProcessID;
      realParameters[1] = parameters;
    }

    for (int i = 0; i < constructors.length; i++)
    {
      // try all the constructors; there really should be only one

      try
      {
        newProgram =
          (Program) constructors[i].newInstance(realParameters);
      }
      catch (Exception ex)
      {
        ex.printStackTrace();
      }
    }

    if (newProgram != null)
    {
      // we created a new process, add it to the table and run it

      Thread newThread = new Thread(newProgram);
      processTable.put(newProcessID, newThread);
      threadTable.put(newThread, newProcessID);
      newThread.start();
      return newProcessID;
    }
    else
    {
      return null;
    }
  }
  catch (Exception e)
  {
    e.printStackTrace();
    return null;
  }
}


/**
 * Creates a new Dynamic UNITY process from the specified Dynamic UNITY
 * program with no instantiation parameters, and returns a Process
 * object identifying it.
 *
 * @param programName The fully qualified Java class name of the class
 *  that implements the Dynamic UNITY program to be instantiated.
```

```
 * @return a Process object identifying the new process, or null if
 *  something went wrong during the instantiation of the new process.
 *
 * @concurrency (GUARDED)
 **/

public synchronized static Process instantiate(String programName)
{
  return instantiate(programName, null);
}


/**
 * Destroys a Dynamic UNITY process by removing it from the process table
 * and notifies the Inbox class of the process's destruction. If the
 * specified process has already been destroyed, this method has no
 * effect.
 *
 * @param The Process object identifying the process to be destroyed.
 **/

static void destroy(Process process)
{
  threadTable.remove(processTable.get(process));
  processTable.remove(process);
  Inbox.removeProcess(process);
}


/**
 * @return an unmodifiable view of the process table. This is for use
 *  primarily by the Inbox class.
 **/

static Map getProcessTable()
{
  return Collections.unmodifiableMap(processTable);
}


/**
 * @return an unmodifiable view of the thread table. This is for use
 *  primarily by the Inbox class.
 **/

static Map getThreadTable()
{
  return Collections.unmodifiableMap(threadTable);
}


/**
 * @return a new unique process ID.
 **/
```

```
private static Process getUniqueProcessID()
{
  Process processID =
    new Process(java.lang.System.currentTimeMillis() +
                "-" + processIDUniquenessValue);

  if (processIDUniquenessValue < Integer.MAX_VALUE - 1)
  {
    processIDUniquenessValue = processIDUniquenessValue + 1;
  }
  else
  {
    processIDUniquenessValue = 0;
  }

  return processID;
}


// Constructor

/**
 * Constructs a new Process object with the specified process identifier.
 *
 * @param id The process identifier.
 **/

private Process(String id)
{
  this.id = id;
}


// Inherited Instance Methods

/**
 * @return a hash code for this object.
 **/

public int hashCode()
{
  return id.hashCode();
}


/**
 * @return true if this object is equivalent to the specified object,
 *  false otherwise.
 **/

public boolean equals(Object object)
{
  if ((object != null) && (object.getClass().equals(this.getClass())))
```

```
    {
      Process otherProcess = (Process) object;

      if (id == null)
      {
        return (otherProcess.id == null);
      }
      else
      {
        return (id.equals(otherProcess.id));
      }
    }
    else
    {
      return false;
    }
  }


  // No Instance Methods
}
```

A.3 Program

Program is the base class for translations of Dynamic UNITY programs, implementing the interface described in Section 7.2.2. This implementation relies on certain features of Java and the message passing system, and on the implementor of its subclasses, to ensure that weak fairness is maintained in the translation of a Dynamic UNITY program. Specifically, it relies on the fact that Java chooses threads for execution from a run queue and so, if all threads contain appropriately-placed `yield()` statements, no thread will ever be starved for execution time. It also relies on the fairness of the message passing system—all sent messages must be delivered to their destinations in finite time assuming that the destinations exist.

CLASS A.3 (*Program*, PART OF A DYNAMIC UNITY RUNTIME FRAMEWORK)

```
package dynamicunity;

import java.io.Serializable;

public abstract class Program implements Runnable
{
  // Instance Variables

  /**
   * A flag indicating whether or not this instantiation of the Program
   * is running.
   **/
```

```
private boolean running;


/**
 * The Outbox used by this instantiation of the Program.
 **/

private Outbox outbox;


/**
 * The Process reference corresponding to this instantiation of the
 * Program.
 **/

private Process process;


// Constructor

public Program(Process process)
{
  this.process = process;
  outbox = new Outbox(process);
  running = true;
}


// Inherited Instance Methods

/**
 * The main run loop of a Dynamic UNITY process.
 **/

public void run()
{
  initialize();

  while (running)
  {
    if (unfairCondition())
    {
      unfairTransition();
    }
    else
    {
      fairTransition();
    }

    Thread.yield();
  }
}
```

```
// Instance Methods

/**
 * Sends a message on the outbox. This implements the Dynamic UNITY
 * send operation.
 **/

protected void send(Process process, String inbox, Serializable message)
{
  outbox.send(process, inbox, message);
}


/**
 * Stops the execution of this instantiation of the Program.
 **/

protected void stop()
{
  running = false;
  Process.destroy(process);
}


/**
 * @return true if this instantiation of the Program is running,
 * and false if it has been stopped.
 **/

public boolean isRunning()
{
  return running;
}


/**
 * @return the Process object associated with this instantiation of the
 * Program.
 **/

public Process getProcess()
{
  return process;
}


/**
 * An abstract method that must be overridden by subclasses to implement
 * initialization. This method should assign initial values to variables,
 * as appropriate.
 **/

protected abstract void initialize();
```

```
  /**
   * An abstract method that must be overridden by subclasses to implement
   * the fair transition set. This method should pick one fair transition
   * in a manner consistent with weak fairness and execute it.
   **/

  protected abstract void fairTransition();


  /**
   * A method with no body that can be overridden by subclasses to
   * implement the selection of unfair transitions. If overridden, this
   * method should select a single unfair transition and execute it.
   **/

  protected void unfairTransition() {}


  /**
   * A method that can be overridden by subclasses to determine the
   * scheduling policy for unfair transitions. This method is called every
   * time through the main run loop. If it returns true, an unfair
   * transition is executed; otherwise, a fair transition is executed. It
   * should therefore return false most of the time. By default, it always
   * returns false (meaning that no unfair transitions are ever executed).
   **/

  protected boolean unfairCondition()
  {
    return false;
  }
}
```

A.4 Outbox

Outbox is the class that handles sending messages within a Dynamic UNITY system. As mentioned previously, we simulate asynchronous message delivery by explicitly placing timestamps on messages. The *Outbox* implementation is responsible for adding these timestamps, and ensures that messages sent from the same *Outbox* instance have monotonically increasing timestamps.

CLASS A.4 (*Outbox*, PART OF A DYNAMIC UNITY RUNTIME FRAMEWORK)

```
package dynamicunity;

import java.io.Serializable;
import java.util.Random;
```

```
class Outbox
{
  // Static Variables

  /**
   * The maximum message delivery delay, in milliseconds.
   **/

  static final int MAX_DELAY = 10000;


  /**
   * The minimum delay between message deliveries, in milliseconds.
   **/

  static final int MIN_DELAY = 10;


  // Instance Variables

  /**
   * The process to which this Outbox belongs.
   **/

  private Process process = null;


  /**
   * The last timestamp that was attached to a message sent by this
   * Outbox.
   **/

  private long lastTimestamp = 0;


  /**
   * The random number generator used by this Outbox.
   **/

  private Random random = new Random();


  // Constructor

  /**
   * Constructs a new Outbox belonging to the specified process.
   *
   * @param process The process.
   **/

  Outbox(Process process)
  {
    this.process = process;
  }
```

```
  // Instance Methods

  /**
   * Sends a message with the specified contents to the specified inbox
   * of the specified process.
   *
   * @param process The destination process.
   * @param inbox The name of the destination inbox.
   * @param message The message contents.
   *
   * @concurrency (GUARDED)
   **/

  synchronized void send
    (Process destination, String inbox, Serializable message)
  {
    int delay = random.nextInt(MAX_DELAY);
    long timestamp =
      Math.max(java.lang.System.currentTimeMillis() + delay,
               lastTimestamp + MIN_DELAY);
    lastTimestamp = timestamp;

    Message packagedMessage = new Message(process, message, timestamp);

    Inbox.deliver(destination, inbox, packagedMessage);
  }
}
```

A.5 Inbox

Inbox is the class that handles message delivery queues in a Dynamic UNITY system. The *Inbox* implementation guarantees that messages are received roughly in order of their timestamps. More precisely, it guarantees that the current message is always the delivered message with the earliest timestamp, but not that there are no undelivered messages with earlier timestamps. This satisfies the first-in first-out requirement for all outbox-inbox pairs, since no outbox ever sends messages with out-of-order timestamps.

CLASS A.5 (*Inbox*, PART OF A DYNAMIC UNITY RUNTIME FRAMEWORK)

```
package dynamicunity;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

```
public class Inbox
{
  // Static Variables

  /**
   * A map from process references to maps from inbox names to Inbox
   * objects.
   **/

  private static Map inboxMapTable =
    Collections.synchronizedMap(new HashMap());


  // Instance Variables

  /**
   * The List of messages in this Inbox.
   **/

  private List messageList = new ArrayList();


  /**
   * The message pointer of this Inbox.
   **/

  private int messagePointer = 0;


  // Static Methods

  /**
   * Delivers a message to the Inbox with the specified name belonging to
   * the specified process. If the specified process exists but does not
   * have an Inbox with the specified name, one is created. If the
   * specified process does not exist in the system, the message is
   * discarded.
   *
   * @param destination The destination process.
   * @param inboxName The destination inbox name.
   * @param message The message.
   **/

  static void deliver
    (Process destination, String inboxName, Message message)
  {
    if (Process.getProcessTable().containsKey(destination))
    {
      Map inboxMap;
      Inbox inbox;

      synchronized (inboxMapTable)
      {
        inboxMap = (Map) inboxMapTable.get(destination);
```

```
      if (inboxMap == null)
      {
        inboxMap = Collections.synchronizedMap(new HashMap());
        inboxMapTable.put(destination, inboxMap);
      }
    }

    synchronized (inboxMap)
    {
      inbox = (Inbox) inboxMap.get(inboxName);

      if (inbox == null)
      {
        inbox = new Inbox(destination, inboxName);
      }
    }

    inbox.enqueue(message);
  }
}


/**
 * Removes a process's inboxes from the system.
 *
 * @param process The process.
 **/

static void removeProcess(Process process)
{
  inboxMapTable.remove(process);
}


// Constructors

/**
 * Constructs a new Inbox with the specified name belonging to the
 * process whose thread calls the constructor. If an Inbox with the
 * specified name already exists for this process, the constructed Inbox
 * is an exact duplicate of it.
 *
 * @param name The inbox name.
 * @exception SecurityException if the thread calling this constructor
 *  is not known to the Dynamic UNITY runtime.
 **/

public Inbox(String name)
  throws IllegalArgumentException
{
  Map threadTable = (Map) Process.getThreadTable();
  Process process = (Process) threadTable.get(Thread.currentThread());
```

```
  if (process == null)
  {
    throw new SecurityException
      ("Inbox constructor called from unknown thread.");
  }

  synchronized (inboxMapTable)
  {
    Map inboxMap = (Map) inboxMapTable.get(process);

    if (inboxMap != null)
    {
      synchronized (inboxMap)
      {
        Inbox oldInbox = (Inbox) inboxMap.get(name);

        if (oldInbox != null)
        {
          this.messageList = oldInbox.messageList;
        }

        inboxMap.put(name, this);
      }
    }
    else
    {
      inboxMap = Collections.synchronizedMap(new HashMap());
      inboxMap.put(name, this);
      inboxMapTable.put(process, inboxMap);
    }
  }
}


/**
 * Constructs a new Inbox with the specified name belonging to the
 * specified process. It is assumed that the process already exists,
 * that no Inbox with the specified name exists for the process, and
 * that the calling thread holds locks for both the inbox map table and
 * the process's inbox map.
 *
 * This method is only called by the Inbox class.
 *
 * @param process The process.
 * @param name The inbox name.
 **/

private Inbox(Process process, String name)
{
  inboxMapTable.put(name, this);
}
```

```
// Instance Methods

/**
 * Enqueues a message in this Inbox. This method is only called by
 * the Inbox class.
 *
 * @param message The message.
 *
 * @concurrency (GUARDED)
 **/

private synchronized void enqueue(Message message)
{
  messageList.add(message);
  Collections.sort(messageList);
}


/**
 * @return true if there is at least one message waiting to be
 *  read from this Inbox.
 *
 * @concurrency (GUARDED)
 **/

public synchronized boolean probe()
{
  while ((messagePointer > 0) && (messageList.size() > 0))
  {
    messageList.remove(0);
    messagePointer = messagePointer - 1;
  }

  return ((messageList.size() > 0) &&
          (((Message) messageList.get(0)).timestamp() <=
           java.lang.System.currentTimeMillis()));
}


/**
 * @return the current message from this Inbox. If there is no
 *  current message (i.e. if probe() returns false), this method
 *  returns null.
 *
 * @concurrency (GUARDED)
 **/

public synchronized Message current()
{
  if (probe())
  {
    return (Message) messageList.get(0);
  }
  else
```

```
    {
      return null;
    }
  }


  /**
   * Advances this Inbox's message pointer.
   *
   * @concurrency (GUARDED)
   **/

  public synchronized void advance()
  {
    if (messageList.size() > 0)
    {
      messageList.remove(0);
    }
    else
    {
      messagePointer = messagePointer + 1;
    }
  }
}
```

A.6 Message

Message is the class that encapsulates a Dynamic UNITY message. It includes methods that allow Dynamic UNITY programs to retrieve a reference to the sender of a message, and to retrieve the contents of the message (as a *Serializable*). The implementation also includes a timestamp, to enable the simulation of asynchronous message delivery within a single virtual machine.

CLASS A.6 (*Message*, PART OF A DYNAMIC UNITY RUNTIME FRAMEWORK)

```
package dynamicunity;

import java.io.Serializable;

public class Message implements Serializable, Comparable
{
  // Instance Variables

  /**
   * The Process that sent this message.
   **/

  private Process process;
```

```
/**
 * The contents of this message.
 **/

private Serializable message;


/**
 * The timestamp of this message.
 **/

private long timestamp;


// Constructor

/**
 * Constructs a new Message with the specified sending process, message
 * contents and timestamp.
 *
 * @param process The sending process.
 * @param message The message contents.
 * @param timestamp The timestamp.
 **/

public Message(Process process, Serializable message, long timestamp)
{
  this.process = process;
  this.message = message;
  this.timestamp = timestamp;
}


// Inherited Instance Methods

/**
 * Compares this object with the specified object for order. Messages are
 * ordered strictly according to their timestamps (their sending
 * processes and contents are not used for ordering).
 *
 * @param object The other object.
 *
 * @exception ClassCastException if the specified object cannot be
 *  compared to this object.
 **/

public int compareTo(Object object)
{
  Message otherMessage = (Message) object;

  if ((timestamp - otherMessage.timestamp) < 0)
```

```
    {
      return -1;
    }
    else if (timestamp == otherMessage.timestamp)
    {
      return 0;
    }
    else
    {
      return 1;
    }
  }


  // Instance Methods


  /**
   * @return the Process object identifying the sender of this Message.
   **/

  public Process process()
  {
    return process;
  }


  /**
   * @return the contents of this Message.
   **/

  public Serializable message()
  {
    return message;
  }


  /**
   * @return the timestamp of this Message.
   **/

  long timestamp()
  {
    return timestamp;
  }
}
```

A.7 Multiset

Multiset is the class that implements the Dynamic UNITY **multiset** data type. It uses the standard Java Collections class *HashMap* to store a mapping from each element of the multiset to the number of instances of that element in the multiset.

CLASS A.7 (*Multiset*, PART OF A DYNAMIC UNITY RUNTIME FRAMEWORK)

```
package dynamicunity;

import java.io.Serializable;
import java.util.HashMap;

public class Multiset implements Serializable
{
  // Instance Variables

  /**
   * A HashMap containing the data members of this Multiset. The keys
   * are the data members, and the values are the number of occurrences
   * of each in the Multiset.
   **/

  private HashMap setMap = new HashMap();


  /**
   * The size of the Multiset. This is cached for performance reasons.
   **/

  private int size = 0;


  // No Constructor

  // Instance Methods


  /**
   * Adds the specified element to the Multiset.
   *
   * @param element The element to add.
   *
   * @concurrency (GUARDED)
   **/

  public synchronized void add(Object element)
  {
    Integer numberInSet = (Integer) setMap.get(element);

    if (numberInSet == null)
```

```
  {
    setMap.put(element, new Integer(1));
  }
  else
  {
    setMap.put(element, new Integer(numberInSet.intValue() + 1));
  }

  size = size + 1;
}


/**
 * Removes one occurrence of the specified element from the Multiset.
 *
 * @param element The element to remove.
 * @return true if an element was removed from the Multiset,
 *  false otherwise.
 *
 * @concurrency (GUARDED)
 **/

public synchronized boolean remove(Object element)
{
  Integer numberInSet = (Integer) setMap.get(element);

  if (numberInSet == null)
  {
    return false;
  }
  else if (numberInSet.intValue() == 1)
  {
    setMap.remove(element);
    return true;
  }
  else
  {
    setMap.put(element, new Integer(numberInSet.intValue() + 1));
    return true;
  }
}


/**
 * Removes all occurences of the specified element from the Multiset.
 *
 * @param element The element to remove.
 * @return the number of occurences removed from the Multiset.
 **/

public synchronized int removeAll(Object element)
{
  Integer numberInSet = (Integer) setMap.get(element);
```

```
    if (numberInSet == null)
    {
      return 0;
    }
    else
    {
      setMap.remove(element);
      size = size - numberInSet.intValue();
      return numberInSet.intValue();
    }
  }


  /**
   * @return true if the Multiset contains the specified element,
   *  false otherwise.
   **/

  public boolean contains(Object element)
  {
    return setMap.containsKey(element);
  }


  /**
   * @return the number of elements in the Multiset.
   **/

  public synchronized int size()
  {
    return size;
  }


  /**
   * @return the number of occurrences of the specified element in the
   *  Multiset.
   **/

  public synchronized int numberOf(Object element)
  {
    Integer numberInSet = (Integer) setMap.get(element);

    if (numberInSet == null)
    {
      return 0;
    }
    else
    {
      return numberInSet.intValue();
    }
  }
}
```

Appendix B

Java Implementation of the Mutual Exclusion Example

The following Java classes implement a translation of the mutual exclusion example from Chapter 5:

B.1 The Resource Program

The Resource program contains only fair, unquantified transitions. Therefore, it is reasonable to implement it using a simple round-robin scheduling approach.

Class B.1 (A translation of the Resource program)

```
package dynamicunity.srme;

import dynamicunity.Inbox;
import dynamicunity.Multiset;
import dynamicunity.Process;
import dynamicunity.Program;

public class Resource extends Program
{
  // Instance Variables

  /**
   * A counter used for iterating through the transitions.
   **/

  private int transitionCounter;


  // Dynamic UNITY Variables

  /**
   * The "requestIn" inbox.
   **/
```

```
protected Inbox requestIn;


/**
 * The "releaseIn" inbox.
 **/

protected Inbox releaseIn;


/**
 * A multiset that holds releases for processing.
 **/

protected Multiset releases;


/**
 * The Process that currently holds the Resource.
 **/

protected Process current;


// Constructor

/**
 * Constructs a new Resource with the specified Process reference.
 *
 * @param process The process reference.
 **/

public Resource(Process process)
{
  super(process);
}


// Instance Methods

/**
 * Initializes the state of the Dynamic UNITY variables.
 **/

public void initialize()
{
  requestIn = new Inbox("requestIn");
  releaseIn = new Inbox("releaseIn");

  releases = new Multiset();
  current = null;
```

```
    transitionCounter = 0;
  }


  /**
   * Implements the fair transition set. This implementation executes
   * the 3 fair transitions in a round-robin fashion.
   **/

  public void fairTransition()
  {
    switch (transitionCounter)
    {
      case 0:
      {
        if ((current == null) && requestIn.probe())
        {
          current = requestIn.current().process();
          requestIn.advance();
          send(current, "tokenIn", null);
        }

        break;
      }

      case 1:
      {
        if ((current != null) && releases.contains(current))
        {
          releases.remove(current);
          current = null;
        }

        break;
      }

      case 2:
      {
        if (releaseIn.probe())
        {
          releases.add(releaseIn.current().process());
          releaseIn.advance();
        }
      }
    }

    transitionCounter = (transitionCounter + 1) % 3;
  }
}
```

B.2 The Client Program

The Client program contains both fair and unfair transitions. While an implementation which simply omits the unfair transitions would be correct, it would not be interesting because it would never request the use of the resource. We therefore want to ensure that the unfair transitions at least have a chance to run. Moreover, we want to ensure that the client holds on to the resource for a variable amount of time, rather than instantly sending it back due to round-robin scheduling of its fair transitions. To accomplish this, we introduce some randomness to the scheduling of transitions. Unfair transitions are scheduled randomly (amongst themselves), with an unfair transition being selected with increasing probability after a random amount of time has passed since the last unfair transition was selected. There are only two fair transitions, so we repeatedly select one a random number of times, and then repeatedly select the other a random number of times, and repeat this cycle. Of course, unfair transitions can execute in between the repeated fair transitions.

CLASS B.2 (A TRANSLATION OF THE CLIENT PROGRAM)

```
package dynamicunity.srme;

import java.io.Serializable;
import java.util.Random;

import dynamicunity.Inbox;
import dynamicunity.Process;
import dynamicunity.Program;

public class Client extends Program
{
  // Instance Variables

  /**
   * The time, in milliseconds past the epoch, at which the Client will
   * release the token if it currently holds one.
   **/

  private long fairTransitionSwitchTime;


  /**
   * The time, in milliseconds past the epoch, before which the Client
   * will not execute an unfair transition.
   **/

  private long unfairTransitionThresholdTime;
```

```
/**
 * The random number generator used to select unfair transitions.
 **/

Random random;


// Dynamic UNITY Variables

/**
 * The Resource process with which this Client communicates.
 **/

protected final Process resource;


/**
 * A flag indicating that this Client is idle.
 **/

protected boolean idle;


/**
 * A flag indicating that this Client is waiting.
 **/

protected boolean waiting;


/**
 * A flag indicating that this Client is busy.
 **/

protected boolean busy;


/**
 * The "tokenIn" inbox.
 **/

protected Inbox tokenIn;


// Constructor

/**
 * Constructs a new Client with the specified Process reference and
 * initialization parameters.
 *
 * @param process The process reference.
 * @param parameters The parameters.
 **/
```

```
public Client(Process process, Serializable[] parameters)
{
  super(process);

  // store parameters

  resource = (Process) parameters[0];
}


// Instance Methods

/**
 * Initializes the state of the Dynamic UNITY variables.
 **/

public void initialize()
{
  tokenIn = new Inbox("tokenIn");

  idle = true;
  waiting = false;
  busy = false;

  random = new Random();
  fairTransitionSwitchTime =
    System.currentTimeMillis() + 1000 + random.nextInt(1190000);
  unfairTransitionNumber = 0;
  unfairTransitionThresholdTime =
    System.currentTimeMillis() + random.nextInt(60000);
}


/**
 * Implements the fair transition set. This implementation uses a time
 * threshold to determine when to execute each transition.
 **/

public void fairTransition()
{
  if (System.currentTimeMillis() < fairTransitionSwitchTime)
  {
    // transition 0

    if (waiting && tokenIn.probe())
    {
      waiting = false;
      busy = true;
      tokenIn.advance();
      System.out.println(getProcess() + " received token, now busy");

      // keep the token at least one second, and at most 2 minutes
```

```
      fairTransitionSwitchTime =
        System.currentTimeMillis() + 1000 + random.nextInt(119000);
    }
  }
  else
  {
    // transition 1

    if (busy)
    {
      busy = false;
      idle = true;
      send(resource, "releaseIn", null);
      System.out.println(getProcess() + " sent token, now idle");
    }

    fairTransitionSwitchTime =
      System.currentTimeMillis() + random.nextInt(60000);
  }
}


/**
 * Implements the unfair transition set. This implementation randomly
 * selects an unfair transition.
 **/

public void unfairTransition()
{
  int unfairTransitionNumber = random.nextInt(4);

  switch (unfairTransitionNumber)
  {
    case 0:
    {
      if (idle)
      {
        idle = false;
        waiting = true;
        send(resource, "requestIn", null);
        System.out.println(getProcess() + " sent request, now waiting");
      }

      break;
    }

    case 1:
    {
      if (idle)
      {
        idle = false;
        System.out.println(getProcess() + " leaving system from idle");
        stop();
      }
```

```
      break;
    }

    case 2:
    {
      if (waiting)
      {
        waiting = false;
        send(resource, "releaseIn", null);

        System.out.println
          (getProcess() + " leaving system from waiting");
        stop();
      }

      break;
    }

    case 3:
    {
      if (busy)
      {
        busy = false;
        send(resource, "releaseIn", null);
        System.out.println(getProcess() + " leaving system from busy");
        stop();
      }
    }
  }

  unfairTransitionThresholdTime =
    System.currentTimeMillis() + random.nextInt(60000);
}


/**
 * Imposes the restriction that unfair transitions can not be executed
 * more often than once per minute.
 **/

public boolean unfairCondition()
{
  // execute an unfair transition with a probability based on the
  // amount of time by which we've exceeded the threshold time;
  // if we're 100 seconds past the threshold time, we'll execute
  // an unfair transition

  long currentTime = System.currentTimeMillis();

  if (currentTime < unfairTransitionThresholdTime)
  {
    return false;
  }
```

```
    long probability =
      (currentTime - unfairTransitionThresholdTime) / 1000;

    if (probability > random.nextInt(100))
    {
      return true;
    }
    else
    {
      return false;
    }
  }
}
```

B.3 The Generator Program

The Generator program contains a single fair transition that creates a new Client process. We use a randomly generated "sleep" period between 10 seconds and 2 minutes to simulate the execution of **skip** transitions so that the Generator doesn't create new Client processes too rapidly. We also keep a parameter list consisting of a single parameter, the resource, as an instance variable. This prevents us from having to construct a new 1-element array every time the fair transition executes.

CLASS B.3 (A TRANSLATION OF THE GENERATOR PROGRAM)

```
package dynamicunity.srme;

import java.io.Serializable;
import java.util.Random;

import dynamicunity.Inbox;
import dynamicunity.Process;
import dynamicunity.Program;

public class Generator extends Program
{
  // Instance Variables

  /**
   * The random number generator used for determining the sleep
   * period between transitions.
   **/

  Random random;

  /**
   * The initialization parameters to be passed to Clients created
   * by this Generator.
   **/
```

```
Serializable[] parameter = new Serializable[1];


// Dynamic UNITY Variables

/**
 * The Resource process created by this Generator.
 **/

protected Process resource;


// Constructor

/**
 * Constructs a new Generator with the specified Process reference.
 **/

public Generator(Process process)
{
  super(process);
}


// Instance Methods

/**
 * Initializes the state of the Dynamic UNITY variables.
 **/

public void initialize()
{
  resource = Process.instantiate("dynamicunity.srme.Resource");
  parameter[0] = resource;
  random = new Random();
}


/**
 * Implements the fair transition set. This implementation repeatedly
 * executes a fair transition and then sleeps for a random period
 * of time.
 **/

public void fairTransition()
{
  Process.instantiate("dynamicunity.srme.Client", parameter);

  try
  {
    Thread.sleep(10000 + random.nextInt(50000));
  }
```

```
    catch (InterruptedException e)
    {
    }
  }
}
```

B.4 The System

The system's initial program is the Generator program. Therefore, the system is translated as follows:

CLASS B.4 (A TRANSLATION OF THE SINGLERESOURCEMUTUALEXCLUSION SYSTEM)

```
package dynamicunity.srme;

import dynamicunity.System;

public class SingleResourceMutualExclusion extends System
{
  // Constructor

  /**
   * Constructs a SingleResourceMutualExclusion system.
   **/

  public SingleResourceMutualExclusion()
  {
    super("Generator", null);
  }


  // main() method

  /**
   * The main() method for SingleResourceMutualExclusion, instantiates
   * a SingleResourceMutualExclusion system.
   **/

  public static void main(String[] argv)
  {
    SingleResourceMutualExclusion srme =
      new SingleResourceMutualExclusion();
  }
}
```

Bibliography

[1] Martin Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15:73–132, January 1993. 140

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, MA, USA, 1986. 141

[3] Flemming Andersen. *A Theorem Prover for UNITY in Higher Order Logic*. PhD thesis, Technical University of Denmark, 1992. 145

[4] J. Dean Brock and William B. Ackerman. Scenarios: A model of non-determinate computation. In *Formalization of Programming Concepts*, volume 107 of *Lecture Notes in Computer Science*, pages 252–259. Springer–Verlag, Heidelberg, Germany, April 1981. 142

[5] K. Mani Chandy, Joseph R. Kiniry, Adam Rifkin, and Daniel M. Zimmerman. Webs of archived distributed computations for asynchronous collaboration. *Journal of Supercomputing*, 11(2):101–118, 1997. 129

[6] K. Mani Chandy, Joseph R. Kiniry, Adam Rifkin, Daniel M. Zimmerman, Wesley Tanaka, and Luke Weisman. A framework for structured distributed object computing. Center for Reserach in Parallel Computing Technical Report CRPC–97–2, California Institute of Technology, February 1997. 129

[7] K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984. 113

[8] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison–Wesley Publishing Company, Reading, MA, USA, 1988. 2, 29, 140

[9] K. Mani Chandy and Beverly A. Sanders. Predicate transformers for reasoning about concurrent computation. *Science of Computer Programming*, 24(2):129–148, April 1995. 42, 140

[10] K. Mani Chandy and Beverly A. Sanders. Predicate transformers for reasoning about concurrent computation (vol 24, pg 129, 1995). *Science of Computer Programming*, 29(3):335, September 1997. Correction. 42, 140

[11] Michel Charpentier and K. Mani Chandy. Towards a compositional approach to the design and verification of distributed systems. In *World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 570–589. Springer–Verlag, September 1999. 3

[12] Michel Charpentier, Mamoun Filali, Philippe Mauran, Gérard Padiou, and Philippe Quéinnec. Tailoring UNITY to distributed program design. In *International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'98)*, volume 1388 of *Lecture Notes in Computer Science*, pages 820–832. Springer–Verlag, April 1998. 142

[13] Michel Charpentier, Mamoun Filali, Philippe Mauran, Gérard Padiou, and Philippe Quéinnec. The observation: an abstract communication mechanism. *Parallel Processing Letters*, 9(3):437–450, 1999. 142

[14] Pierre Collette. Composition of assumption–commitment specification in a UNITY style. *Science of Computer Programming*, 23(2–3):107–125, 1994. 140

[15] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Development System*. Prentice–Hall, Inc., Upper Saddle River, NJ, USA, 1986. 145

[16] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice–Hall, Inc., Upper Saddle River, NJ, USA, 1976. 139

[17] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer–Verlag, Heidelberg, Germany, 1990. 30, 139

[18] distributed.net. `http://www.distributed.net/`, 1997. 1

[19] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Mathematics*, volume 19, pages 19–31, 1967. 139

[20] David Gelertner. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7:80–112, January 1985. 142

[21] David Gelertner and A. Bernstein. Distributed communications via global buffer. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 10–18, Ottawa, Canada, August 1982. 142

[22] M. J. C. Gordon and T. F. Melham. *Introduction to HOL—A theorem proving environment for higher order logic*. Cambridge University Press, Cambridge, England, 1993. 145

[23] Eric C. R. Hehner. *A Practical Theory of Programming*. Springer–Verlag, Heidelberg, Germany, 1993. 3, 10

[24] C. E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, June 1977. 141

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969. 30, 139

[26] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978. 14, 30, 139, 142

[27] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice–Hall, Inc., Upper Saddle River, NJ, USA, 1984. 142

[28] D.R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985. 114

[29] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983. 140

[30] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification: Second Edition*. Addison–Wesley Publishing Company, Reading, MA, USA, 2000. 128

[31] Fred Kröger. *Temporal Logic of Programs*, volume 8 of *EATCS Monographs on Theoretical Computer Science*. Springer–Verlag, Heidelberg, Germany, 1987. 140

[32] S. S. Lam and A. U. Shankar. A theory of interfaces and modules 1: Composition theorem. *IEEE Transactions on Software Engineering*, 20:55–71, January 1994. 140

[33] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. 114

[34] Leslie Lamport. win and sin: Predicate transformers for concurrency. Technical Report SRC-017, Digital Systems Research Center, Palo Alto, California, May 1987. Revised December 1989. 139

[35] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16:872–923, May 1994. 3, 10, 140

[36] Leslie Lamport. Specifying concurrent systems with TLA+. In *Calculational System Design*. IOS Press, Amsterdam, The Netherlands, 1999. 140

[37] K. Rustan M. Leino. *Toward reliable modular programs*. PhD thesis, Department of Computer Science, California Institute of Technology, 1995. 29

[38] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1996. 142

[39] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2:219–246, September 1989. 142

[40] Zohar Manna and Amir Pneuli. *Temporal Verification of Reactive Systems*. Springer–Verlag, Heidelberg, Germany, 1995. 140

[41] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer–Verlag, Heidelberg, Germany, 1992. 140

[42] Alain J. Martin. An axiomatic definition of synchronization primitives. *Acta Informatica*, 16:219–235, October 1981. 139

[43] Robin Milner. *Communication and Concurrency*. Prentice–Hall, Inc., Upper Saddle River, NJ, USA, 1989. 141

[44] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. I. *Information and Computation*, 100(1):1–40, September 1992. 141

[45] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. II. *Information and Computation*, 100(1):41–77, September 1992. 141

[46] Jayadev Misra. A logic for concurrent programming: Progress. *Journal of Computer & Software Engineering*, 3(2):273–300, 1995. 43, 44, 47, 48

[47] Jayadev Misra. A logic for concurrent programming: Safety. *Journal of Computer & Software Engineering*, 3(2):239–272, 1995. 42, 47, 141

[48] Jayadev Misra. A discipline of multiprogramming. *ACM Computing Surveys*, 28, December 1996. 141

[49] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, July 1981. 142

[50] P. Mockapetris. RFC 1034: Domain Names—Concepts and Facilities. `http://www.ietf.org/rfs/rfc1034.txt`. 1

[51] P. Mockapetris. RFC 1034: Domain Names—Implementation and Specification. `http://www.ietf.org/rfs/rfc1035.txt`. 1

[52] Carroll Morgan. *Programming from Specifications*. Prentice–Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1994. 30

[53] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. I. *Acta Informatica*, 6:319–340, 1976. 139

[54] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19:279–285, May 1976. 139

[55] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer–Verlag, Heidelberg, Germany, 1994. 145

[56] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. LIME: Linda meets mobility. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*. ACM Press, May 1999. 142

[57] Gian Pietro Picco, Gruia-Catalin Roman, and Peter J. McCann. Reasoning about code mobility with Mobile UNITY. Technical Report WUCS–97–43, Department of Computer Science, Washington University, December 1997. 141

[58] Amir Pneuli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*, pages 46–57. IEEE Press, 1977. 140

[59] Jonathan B. Postel. RFC 821: Simple Mail Transport Protocol. `http://www.ietf.org/rfc/rfc821.txt`. 1

[60] Arthur N. Prior. Time and Modality. In *John Locke Lectures, 1955-6*. Clarendon Press, Oxford, 1957. 140

[61] The LogiCal Project. The Coq Proof Assistant. `http://coq.inria.fr/`. 145

[62] Nicholas Rescher and Alasdair Urquhart. *Temporal Logic*, volume 3 of *Library of Exact Philosophy*. Springer–Verlag, Heidelberg, Germany, 1971. 42, 140

[63] Gruia-Catalin Roman, Peter J. McCann, and Jerome Y. Plun. Mobile UNITY: Reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, July 1997. 141

[64] Richard D. Schlicting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983. 143

[65] SETI@Home: The Search for Extraterrestrial Intelligence. `http://setiathome.ssl.berkeley.edu/`, 1999. 1

[66] Paolo A. G. Sivilotti. *A Method for the Specification, Composition, and Testing of Distributed Object Systems*. PhD thesis, Department of Computer Science, California Institute of Technology, 1997. 42, 43, 44, 49, 140, 142

[67] Sun Microsystems, Inc. *Java Remote Method Invocation Specification*. Sun Microsystems, Inc., 1.7 edition, December 1999. 129

[68] Daniel M. Zimmerman. A preliminary investigation into dynamic distributed workflow. Master's thesis, Department of Computer Science, California Institute of Technology, 1998. 129

Index