A UNITY-based Formalism for Dynamic Distributed Systems

Daniel M. Zimmerman
Computer Science 256-80
California Institute of Technology
Pasadena, California 91125 USA
dmz@cs.caltech.edu

Abstract

We describe Dynamic UNITY, a new formalism for the specification of dynamic distributed systems based on the UNITY formalism. This formalism allows for the specification and proof of systems where processes may be created and destroyed, and where communication links among processes may change. It also introduces asynchronous messaging as a primitive construct, to facilitate the composition of multiple programs into a larger system.

We also present an example Dynamic UNITY system that illustrates the dynamic aspects of the new formalism, and outline a correctness proof for the example.

Keywords: *dynamic distributed systems, formal methods, program composition, UNITY*

1. Introduction

Distributed systems, or systems that consist of multiple communicating processes, are becoming both increasingly common and increasingly important. A distributed system may be either *static*, meaning that it is comprised of a fixed set of processes with fixed communication links among them, or *dynamic*, meaning that its component processes may be created or destroyed during the system's execution and that the communication links among its component processes may change over time.

Several well-known and well-understood formal methods, such as Chandy and Misra's UNITY formalism [2], Lamport's TLA [8] and TLA+ [9], and Lynch and Tuttle's I/O Automata [10, 11], exist for specifying static distributed systems and proving their correctness. Dynamic aspects have been introduced in multiple variants of the UNITY language and logic: for example, Cunningham and Roman's Swarm [6], a UNITY-based formal method for shared dataspaces, introduces dynamic transaction sets, while Roman's Mobile UNITY [14], an extension of UNITY to systems of mobile agents, introduces dynamic communication patterns among processes in a system. However, there are few formal methods that specifically address the issues inherent in dynamic distributed systems; Attie and Lynch's Dynamic I/O Automata [1], an extension of I/O Automata to dynamic systems, is one of these.

This work describes a UNITY-based formalism, called Dynamic UNITY, that can be used for the specification and proof of dynamic distributed systems. The operators of the UNITY logic are left essentially unchanged in Dynamic UNITY, despite changes to the underlying execution model, so that proof techniques developed for UNITY may be applied to Dynamic UNITY systems. The Dynamic UNITY language, however, differs significantly from the UNITY language.

In addition to describing the Dynamic UNITY formalism, we present an example Dynamic UNITY system that illustrates the dynamic aspects of the new formalism and briefly outline a proof of the example system's correctness.

Paper outline. The paper begins with a brief overview of UNITY and its limitations with respect to dynamic systems (Section 2). We then discuss the changes necessary to extend UNITY to handle dynamic systems, and provide an overview of the Dynamic UNITY language and logic (Section 3). Finally, we present an example system (Section 4) and conclusions (Section 5).

2. The UNITY formalism

The UNITY formalism consists of both a programming language (with accompanying execution model) and a proof logic. We briefly describe both, as a basis for discussion of our new formalism.

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

```
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
```

Figure 1. A UNITY program that implements integer division.

2.1. Language

In the UNITY programming language, a program consists of a set of state variables, a set of initialization assignment statements, and a set of guarded multiple assignment statements (the *transitions*). The set of transitions always includes **skip**, the transition that changes no part of the state. An example UNITY program that divides M by N is shown in Figure 1.

2.2. Execution model

Execution of a UNITY program proceeds in the following way. First, the initialization statements are executed to set the state variables to their initial values. Then, transitions are repeatedly chosen and executed atomically. Transition selection is subject to a weak fairness constraint, which insures that every transition is executed infinitely often in every infinite execution of the program.

2.3. Logic

The UNITY logic [2, 12, 13] is a temporal logic based on three operators: **next**, **transient**, and **initially**. The fundamental operator for safety is **next**[1]: p **next** q means that all transitions from states where p holds are to states where q holds. The fundamental operator for progress is **transient**: **transient** p means that there exists a transition that takes the system from any state where p holds to a state in which $\neg p$ holds. An additional operator, **initially**, is also used: **initially** p means that predicate p holds in all initial states of the system.

[1] Misra writes **co**, "constrains," instead of **next**.

The fundamental operators are used to define other useful operators, such as safety operators **stable** and **invariant**, the progress operator $\leadsto$ (leads-to), and operators such as Sivilotti's **follows** [15] and Charpentier's $\preccurlyeq$ (observation) [5] that combine both safety and progress. There are two forms of the fundamental operators: the weak form, defined using initial system states and induction over the set of transitions, considers only reachable states; the strong form considers all possible states regardless of their reachability.

2.4. Modelling dynamic systems

UNITY is a useful formalism, primarily because of its simple language and execution model. The weak fairness constraint, the lack of a sequencing operator, and the fact that multiple assignment statements are well understood constructs help to make UNITY programs straightforward to analyze.

However, because UNITY programs are static sets of state variables and transitions, it is inconvenient to describe systems that exhibit dynamic behavior with UNITY. If all the possible system behaviors are known at the time a UNITY program is written (a situation where the system behavior is arguably static), then transitions can be written to encompass every one of those possibilities. If all the possible behaviors of a system are not known in advance, UNITY is inadequate to the task of describing the system.

UNITY programs are difficult to compose into multiple-program systems, because UNITY does not have the concept of processes. This difficulty is especially problematic when describing dynamic systems. In order to reason about a system that exhibits dynamic behavior with respect to process creation, process destruction, and inter-process communication, the programs underlying individual processes must be considered as composable units because it is impossible to construct *a priori* a large static program that models the system's behavior.

3. The Dynamic UNITY formalism

To adapt UNITY to the specification and proof of dynamic distributed systems, we introduce changes to the programming language, the proof logic, and the execution model. To minimize the extent of these changes, 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, using a model in which each process may

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

have multiple named inboxes and may address messages to any named inbox belonging to any process it knows.

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

While these constraints may prevent us from being able to model certain types of dynamic behavior, they do allow for many behaviors that can not be modeled with UNITY alone. In addition, they also help to simplify the tasks of programming and proving the correctness of dynamic systems. The first constraint eliminates any possibility that a process can directly interfere with the operation of another process, enabling both modular reasoning (proof reuse) and modular system construction. The second constraint restricts interprocess communication to a single well-understood mechanism, which simplifies system design. The third constraint simplifies proof obligations by eliminating the possibility that a running process will be destroyed at an unexpected or inappropriate point of its execution. The combination of all three constraints allows us to prove properties about individual programs independently of the other programs in the system.

Each constraint mandates specific changes to the UNITY formalism. We introduce processes, instantiations of programs that are able to create other processes and halt their own executions, and we eliminate shared variables among processes. We add reliable first-in first-out message passing to the language via primitives that manipulate messages and inboxes in various ways. We include the ability to create transitions that are not subject to fairness constraints, which enables more accurate modelling of distributed systems in which particular events (such as requests in a resource allocation system) may not occur in a fair manner. We also introduce the ability to quantify transitions over process state, which enables the set of transitions within a single process to change as its state changes.

3.1. Language

The new primitives added to Dynamic UNITY for message passing and process manipulation are as follows:

- **send** is used to send one or more messages.
- **probe** is used to determine whether there is an available message on a particular inbox.
- **current** is used to access the current message on a particular inbox.
- **advance** is used to advance to the next message on a particular inbox.
- **name** is used to access, read-only, the name of a particular inbox.
- **type** is used to determine the type of a received message.
- **length** is used to determine the length of an entity (such as a string or a list).
- **this** is used to obtain a reference to the current process, which can then be sent as a message to other processes in order to establish communication links.
- **new** is used to create a new process.
- **stop** is used by a process to halt its own execution (destroy itself).

Rather than trying to frame these new primitives as assignment statements we change the program notation to one based on binary predicates, similar to the notation of Hehner [7] and of Lamport's TLA [8]. This makes Dynamic UNITY more expressive than UNITY; any UNITY program can be transformed into a Dynamic UNITY system without changing its semantics, but it is possible to construct Dynamic UNITY systems, even without using any of the new primitives, that cannot be transformed into UNITY programs.

We do not describe the syntax of the Dynamic UNITY language in detail here; a grammar for Dynamic UNITY, as well as more precise definitions of the new primitives, can be found in the author's dissertation [16].

Figure 2 shows a Dynamic UNITY program that implements the same division algorithm as the UNITY program of Figure 1; when instantiated it performs the calculation for a specified M and N, sends the result as two messages to a specified destination, and destroys itself. Each of the three assignment statements from the original UNITY program is translated directly into the new notation. The guard for the transition that sends the result and destroys the process is exactly the predicate that holds at all fixed points of the original program.

3.2. Execution model

The execution model of Dynamic UNITY is similar to that of UNITY, in that both atomically change the state of the system by executing a single transition at a time from a transition set; however, there are significant differences between the two.

The most important difference relates to the transition sets themselves: in a UNITY system the transition set is statically determined by the program text, and all transitions are subject to a fairness constraint; in a Dynamic UNITY system the transition set is dynamically determined by the

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

Figure 2. A Dynamic UNITY program that implements integer division.

system state, and only a subset of the transitions need be subject to a fairness constraint. When a program is instantiated as a process, its transitions are added to the system transition set, and when the process terminates itself, its transitions are removed from the system transition set.

A transition in a Dynamic UNITY system is an instantiation of a transition statement within a Dynamic UNITY program, distinguished by the process to which it belongs and any quantifying terms used to generate it. Therefore, different instantiations of the same Dynamic UNITY program have different transition sets. Moreover, a quantified transition is considered not as a single transition, but as a set of transitions determined by the range of the quantification.

Transitions are classified as either weakly fair or unfair, depending on where they occur within the program text. We define weak fairness for Dynamic UNITY as follows: *In every computation of a Dynamic UNITY system, every weakly fair transition is infinitely often either executed or not present in the system.* This definition implies that a transition that remains in the system forever will be executed infinitely often, but it does not imply that a transition that is merely present in the system infinitely often will execute infinitely often.

There are no guarantees with respect to the execution of unfair transitions; in an infinite execution, a single unfair transition may not be executed at all, may be executed only a finite number of times, or may be executed an infinite number of times.

The two execution models also differ in their treatment of the state space. In UNITY, a program's state is comprised of exactly the variables declared in the program text. In Dynamic UNITY, a process's state contains not only the variables declared in its program text, but also the portion of the message passing system state associated with that process. The message passing state for a process contains a sequence of messages for its outbox and sequences for each of its inboxes, as well as additional information indicating which messages in the outbox have been delivered and how many messages have been read from each inbox. Some of this state can be implicitly accessed and changed by executing message passing primitives, but some of it can not be modified by the process. The portion of the state that can be modified by the process, either directly or through message passing primitives, is called the *non-volatile state*; the portion that can only be modified by the message passing system is called the *volatile state*. As will be described in more detail later, this distinction is what enables us to prove properties for every instance of a particular Dynamic UNITY program regardless of its environment.

Every Dynamic UNITY system has an *initial program*, which is always the first program instantiated when the system is executed. At each system step, at most one transition is executed. The execution of a transition belonging to a particular process may result in any or all of the following: changes to the non-volatile state of the process, the creation of one or more new processes, and the destruction of the process. The volatile state of the process may only be changed "in between" transitions, by the message passing system.

We do not include a formal definition of Dynamic UNITY's execution model here due to space constraints; however, one is available in the author's dissertation [16].

3.3. Logic

The Dynamic UNITY logic is a temporal logic capable of expressing the same properties as the UNITY logic; how-

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

ever, because of the introduction of dynamic transition sets and the elimination of shared variables, the fundamental operators are defined and used in a different way. The **initially**, **next**, and **transient** operators are defined in terms of the formal Dynamic UNITY execution model. Just as in UNITY, these properties form a basis for the rest of the logic.

The **next** and **transient** properties cannot be proven directly for Dynamic UNITY systems; since they are defined in terms of executions, doing so would require verifying the properties for every possible execution. However, we can prove stronger properties about systems, or subsets thereof (including individual programs), similar to the *existential* properties of Chandy and Sanders [3, 4]. A property p of a subsystem X is an existential property if and only if p holds for all systems containing X. In other words, an existential property of subsystem X holds for a system consisting of only X, as well as for a system consisting of X composed with additional processes $Y_1, Y_2, \ldots, Y_n$, regardless of what those processes are.

We can use existential properties of a subsystem to infer **next** and **transient** properties of the subsystem. This is straightforward if the properties only depend on non-volatile state, because the non-volatile state of a subsystem can only be changed by a transition within that subsystem. For instance, if we have a subsystem X and predicates p and q over the non-volatile state of X, and we can show that every possible transition from a state satisfying p establishes a state satisfying q, then we can infer (p **next** q).X. Similarly, if we can show that there exists a single fair transition that establishes a state satisfying $\neg p$ from any state satisfying p, then we can infer (**transient** p).X. If we choose our subsystem X to be *a single instance of program* P, all the possible transitions are contained within P's program text. Therefore, we can prove existential properties of P using only its program text, if we restrict these properties to non-volatile state.

To prove properties that depend on volatile state (that is, those properties that involve communication among processes), we reason about the behavior of message channels. This behavior is governed by the Channel Theorem, which states that the following safety and progress properties hold for every process/inbox pair in a system: *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 *is a prefix of the sequence of messages sent by process* p *to* q.b, *and every message sent by process* p *to* q.b *is eventually delivered*. The Channel Theorem is expressed with a single **follows** property that combines these safety and progress conditions. This effectively creates an existential property for each process/inbox pair in a system, which can be combined with properties on non-volatile state to prove properties about interprocess interactions.

4. An example system

To illustrate the use of Dynamic UNITY, we consider the problem of guaranteeing mutually exclusive access to a shared resource in a dynamic environment. We require that only one client at a time has access to the resource, and that no client that requests the resource is forced to wait forever to access it. Clients are allowed to enter and leave the system at any point, but the resource is guaranteed to remain in the system forever. We present a Dynamic UNITY system that solves this problem, and outline some details of a correctness proof (the full proof can be found in the author's dissertation).

There are two main process types in the system, representing the resource and its clients. A third process type, which bootstraps the system by creating a resource process and client processes, is also necessary for the model (in a real system, client processes would be created by external agents such as users).

We choose a simple mutual exclusion algorithm: there is a single token in the system, and only a client that holds the token may use the shared resource. We require that a client that holds the token does not do so forever. Possible states for a client are *idle*, *waiting* (having requested the resource), and *busy* (using the resource); possible states for the resource are *idle* (holding the token, and therefore not being accessed by a client) and *busy* (not holding the token, and therefore potentially being accessed by a client). Valid state transitions for clients are from idle to waiting, from waiting to busy, and from busy to idle, and valid state transitions for the resource are from idle to busy and from busy to idle.

Conceptually, three types of message are sent in the system: requests, releases, and permissions. Requests are sent from clients that want access to the resource, releases are sent by clients that are either done using the resource or are leaving the system (when a request is outstanding), and permissions are sent by the resource to clients.

4.1. Resource

A resource (Figure 3) starts in the idle state. It listens for requests on one inbox (*requestIn*), and releases on another inbox (*releaseIn*). Requests are handled in the order in which they are received, which helps to fulfill our global fairness constraint. Releases, which are tracked using a multiset[2], effectively cancel requests. When the resource is idle and there is a request from a client at the head of the request queue, it is handled in one of two ways: if the resource already holds a release from that client, the request is

[2]The multiset allows a resource to hold two identical releases simultaneously, as no information other than a client reference is conveyed in a release.

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

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 $= \emptyset$

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

end

Figure 3. The "Resource" 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

Figure 4. The "Client" program.

cancelled and the release is removed from the multiset; otherwise, a permission is sent to the client and the resource transitions to busy. A busy resource transitions to idle when it holds a release from the client to which it most recently sent a permission; at that point, the release is removed from the multiset.

We can prove many existential properties about the behavior of a resource that hold independently of the behavior of other processes. Some are counting properties, such as that the number of permissions it has sent is equal to the number of requests it has read. Others are state transition properties, such as that a resource never sends two permissions without becoming idle in between and that an idle resource will eventually become busy if there is a request on its inbox.

4.2. Client

A client (Figure 4) starts in the idle state. It sends a request when it transitions from idle to waiting, transitions from waiting to busy when it receives a permission, and sends a release when it transitions from busy to idle; it also sends a release when it leaves the system if it is in the waiting or busy state.

As with the resource, we can prove many existential properties about the behavior of a client that hold independently of the behavior of other processes. Some are counting properties, such as that the number of requests a client has sent is always at most one greater than the number of releases it has sent. Others are state transition properties, such as that the client never transitions from the idle state to the busy state directly and that the client never remains in the busy state forever.

4.3. Generator

The generator is specified as part of the composed system (Figure 5). It is the system's initial program, and its role is straightforward: it creates the resource, and then repeatedly creates clients that know how to contact the resource. We can prove existential properties about it independently of the behavior of other processes, such as that all the clients it creates have references to the same resource process and that it only creates a single resource process.

4.4. The composed system

Once we have proven properties about the processes in isolation, we can use properties about the Dynamic UNITY message passing system to prove the correctness of the composed system. In this system the required safety condition is that there is always exactly one token in the system, and the required progress condition is that every waiting client eventually either transitions to busy or leaves the system.

```
system SingleResourceMutualExclusion

  initial-program Generator

    declare
      resource: process

    initially
      resource = new Resource

    unfair-transition
      (1)  p: p′ = new Client(resource)

  end

  program Resource

  program Client(resource: process)

end
```

Figure 5. The "SingleResourceMutualExclusion" system.

To prove the safety condition, we introduce the concept of "live" tokens. 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, and the number of live tokens in transit. A live token in transit from a client to the resource is a release message with a corresponding request that has been handled by the resource. A live token in transit from the resource to a client is a permission message with a destination client that has not left the system. We can show that having exactly one live token in the system satisfies the safety condition and then prove, using the previously proven properties of the individual programs, that there is exactly one live token in the system at all times.

To prove the progress condition, we observe that every request message sent to the resource must eventually arrive, and that once a request is enqueued in the resource's inbox, no other request can overtake it. Since no client may stay in the busy state forever, and an idle resource must eventually service a request if one arrives, this is sufficient to show progress.

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

5. Conclusion

Distributed systems are often modelled as static multigraphs, where the nodes represent processes and the edges represent communication channels. Such a model is insufficient for dynamic distributed systems, such as electronic marketplaces and artificial life systems. These systems can be modelled as a set of rules that determines how processes are created and destroyed, as well as how they discover each other and interact; in some cases, this set of rules may be allowed to evolve in response to system events. Often, dynamic systems are studied from an observational and behavioral viewpoint. The work described in this paper is a small step toward developing formal methods for reasoning about such systems.

This paper has presented a formalism, based on the familiar **next**, **transient**, and **initially** operators used in UNITY, applicable to dynamic distributed systems. Although our rules for the creation and interaction of processes are not completely general—for example, we prohibited processes from terminating other processes—we have shown that proof methods for static distributed systems can be employed for some dynamic distributed systems.

Acknowledgements

The author would like to thank K. Mani Chandy for his valuable comments and suggestions.

This research was 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.

References

[1] P. C. Attie and N. A. Lynch. Dynamic input/output automata: A formal model for dynamic systems. In *International Conference on Concurrency Theory*, volume 2154 of *Lecture Notes in Computer Science*, pages 137–151. Springer–Verlag, Heidelberg, Germany, Aug. 2001.

[2] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison–Wesley Publishing Company, Reading, MA, USA, 1988.

[3] K. M. Chandy and B. A. Sanders. Predicate transformers for reasoning about concurrent computation. *Sci. Comput. Programming*, 24(2):129–148, Apr. 1995.

[4] K. M. Chandy and B. A. Sanders. Reasoning about program composition. CISE Technical Report 2000-003, University of Florida, 2000.

[5] M. Charpentier, M. Filali, P. Mauran, G. Padiou, and P. Quéinnec. The observation: an abstract communication mechanism. *Parallel Processing Letters*, 9(3):437–450, 1999.

[6] H. C. Cunningham and G.-C. Roman. A UNITY-style programming logic for a shared dataspace language. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):365–376, July 1990.

[7] E. C. R. Hehner. *A Practical Theory of Programming*. Springer–Verlag, Heidelberg, Germany, 1993.

[8] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16:872–923, May 1994.

[9] L. Lamport. Specifying concurrent systems with TLA+. In *Calculational System Design*. IOS Press, Amsterdam, The Netherlands, 1999.

[10] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1996.

[11] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2:219–246, Sept. 1989.

[12] J. Misra. A logic for concurrent programming: Progress. *Journal of Computer & Software Engineering*, 3(2):273–300, 1995.

[13] J. Misra. A logic for concurrent programming: Safety. *Journal of Computer & Software Engineering*, 3(2):239–272, 1995.

[14] G.-C. Roman, P. J. McCann, and J. Y. Plun. Mobile UNITY: Reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, July 1997.

[15] P. 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.

[16] D. M. Zimmerman. *Dynamic UNITY*. PhD thesis, Department of Computer Science, California Institute of Technology, 2001.