Real-Time Unity

Al Carruth
Department of Computer Sciences
Taylor Hall 2.124
University of Texas at Austin
Austin TX 78712-1188
USA

e-mail: carruth@cs.utexas.edu

March 29, 1994

Abstract

We propose Real-Time Unity in which the Unity operators *co* and $\mapsto$ are generalized to the bounded forms co_k and $\mapsto_k$, where k is a time value. This is done in such a way that for $k = \infty$ the bounded forms specialize to the unbounded forms. Hence Real-Time Unity includes Unity as a sub-theory. Real-Time Unity appears to be especially appropriate for reasoning about the interplay of real-time progress and safety properties. We argue that this sort of interplay is fundamental to the development of real-time programs and give a number of examples of the application of the theory to programs which require such an interplay. We then propose topics for further research.

Keywords: Real-time, Unity, program refinement, concurrency.

1 Introduction

The problem of specifying and proving properties of real-time programs has received much attention over the last ten years. This attention is due to the critical function which many of these programs provide. Since many real-time programs are so critical, it is important that we gain the ability to reason effectively about their correctness.

Real-time constraints arise in a variety of ways. Many program specifications include some hard real-time requirement so that the system may properly interact with its physical environment. Common examples include control systems for nuclear power plants, weapons guidance systems or flight control systems. Other real-time constraints may be introduced during the refinement of an untimed specification. For example, the top level specification of a communications protocol might only require the successful transmission of the data while the sub-specifications might require the sender to go slowly and the receiver to go quickly. Some example problems we will consider are:

- **Simple "ping" with a timeout.** One process attempts to determine whether another is up or not. After waiting for some amount of time the process can safely assume the other is not up on the grounds that it would have responded by now.

- **Bounded buffer with asynchronous communication.** Long distance communication can be expedited if the sender can send packets at regular intervals rather than waiting after each packet for the receiver to signal that it is ready to receive the next packet. If the receiver is removing the packets more quickly than the sender is sending them, then they will not be over-written by the next packet before they are read.

- **Fischer's mutual exclusion protocol.** Fischer's protocol guarantees mutual exclusive access to a critical section for at least one of the processes attempting to enter the critical section. The correctness of this protocol depends on bounded progress and safety constraints imposed on the processes.

- **Window of opportunity.** A process can transit from state a to b only during a certain window of opportunity which is repeatedly open and closed. If the window is guaranteed to remain open for time k and the process is guaranteed to act within time k when the window becomes open, then the process will eventually transit from a to b.

All of these problems have in common the composition of a *real-time safety property* and a *real-time progress property.*[1] For the program to be correct, one process must go quickly and another must go slowly. To formalize these notions, we propose a quantitative extension of the Unity logic. By this we mean that the usual operators of Unity are generalized by the addition of a time parameter. For instance, $p \;\mapsto\; q$ is generalized to $p \;\mapsto_k\; q$, which means whenever p holds q will hold within time k. We do this in such a way that instantiating k with ∞ specializes the general form to the old untimed form. Likewise many of the usual Unity inference rules have general real-time counterparts in our theory. As a result, proofs in our theory have much the same form and flavor as Unity proofs. We are encouraged by the fact that many of the example problems we have investigated have simple efficient proofs in our theory.

The remainder of the paper is organized as follows. In section 2 we provide the motivation for our approach. In section 3 we give an overview of the untimed Unity logic as it occurs in [Mis93]. In section 4 we present the Real-Time Unity

[1]What we call a real-time progress property others would call a safety property. Indeed many researchers regard all real-time properties as safety properties [AL92], [SBM92]. If one defines safety properties as those properties whose failure for a computation can be detected in a finite prefix of that computation, then indeed all real-time properties are safety properties. Our justification for calling certain real-time properties progress properties lies in the way they behave logically. Because our rules for $p \mapsto_k q$ are analogous to the rules for the untimed $p \mapsto q$, we refer to the former (as well as the latter) as a progress property.

logic. This consists of a syntax for programs and their properties and a number of axioms and inference rules for proving that properties hold for a program. In section 5 we show how the Real-Time Unity logic can be applied to some of the example problems of section 1. In section 6 we present a refinement rule for Real-Time Unity programs. We show how this rule can be used to avoid global state sharing by introducing timing constraints. In section 7 we discuss related approaches to real-time. Finally, in section 8 we propose a future course of work.

2 Why Real-Time Unity?

The motivation for the development of Real-Time Unity as a way to reason about real-time computation is similar to that for Unity with respect to untimed computation. We intend to demonstrate that Real-Time Unity is to bounded operator linear temporal logic as Unity is to linear temporal logic. That is, Real-Time Unity formalizes a subset of bounded operator linear temporal logic that seems to be especially appropriate for the design and verification of real time systems, just as Unity formalizes a subset of linear temporal logic that is especially appropriate for the design and verification of untimed systems.

2.1 Unity vis-à-vis linear temporal logic

Since its application to programming was proposed by Pnueli [Pnu77], temporal logic has become widely accepted as a way to reason about computation, especially for parallel and reactive systems. Unity is just a subset of ordinary linear temporal logic, and hence is not as general. But Unity is a deductive system intended for use by people rather than machines. The appropriateness of the operators and the usefulness of the inference rules are of paramount importance. The notions of *progress* (a.k.a. *liveness*) and *safety* have been identified as fundamental in the study of reactive and concurrent computing systems. By design, these notions are captured directly in Unity by the fundamental operators $\mapsto$ and *co* ($\mapsto$ and *unless* in the original Unity theory). Of course these operators can be expressed in more general temporal logics, and Unity sacrifices some of the expressibility of these other logics. However, we feel that it is more appropriate to judge a theory such as Unity by its successful application to a large problem set rather than by some abstract notion of expressibility. The original Unity text by Chandy and Misra [CM88] provides convincing empirical evidence that Unity is applicable to a large class of problems.

2.2 Real-time extensions to temporal logic

Many researchers have extended temporal logic to reason about real-time. We focus here on two approaches: the *explicit time variable* approach and the *bounded operator* approach. In the explicit time variable approach, the logic itself is not extended. Rather, a variable to denote the current time, say *now*, is introduced and axiomatized within the theory. This allows considerable expressive power. For instance, bounded response, the property that each occurrence of p is followed by an occurrence of q within time k can be formalized as follows:

$$(0) \qquad (\forall t :: \Box(p \wedge now = t \Rightarrow \Diamond(q \wedge now < t + k)))$$

Bounded response is a very important and fundamental concept in real-time programming. If it is this cumbersome to express bounded response, one can imagine the difficulty with which more complex properties and inference rules might be expressed. A more concise (but less general) notation is provided by using bounded operators. In this approach one may write, for example, $\Box_{[j,k]}p$ to indicate that p is always true in the interval extending from j to k time units from now. Similarly one may write $\Diamond_{[j,k]}p$ to indicate that p will be true sometime in the same interval. Hence one can express bounded response as follows.

$$(1) \qquad \Box(p \Rightarrow \Diamond_{[0,k]}q)$$

The formula in (1) is a great improvement over that in (0). Nevertheless, one may hope for something better. Rather than using the interval $[j,k]$, perhaps an upper bound would suffice. And if bounded response seems so important, perhaps we should not require the use of three operators to express it. We suggest (2), which uses a single ternary operator to express bounded response. Note that this is just a bounded form of the Unity leads-to operator.

$$(2) \qquad p \;\mapsto_k\; q$$

The story is similar for safety. We take our cue from Unity and suspect that some bounded form of *co* may be appropriate. Essentially $p\ \ co_k\ \ q$ means that whenever p becomes true, $p\ \ co\ \ q$ holds for at least time k. The reader may wish to see for themselves how difficult it is to express this in the bounded operator linear temporal logic. It turns out that the bounded version of *co* is indeed an important notion which can be used to express the waiting a process does before it times out.[2]

In the following sections we will formalize these concepts, as well as some others, and present a number of inference rules analogous to those in Unity. The task then is to see if the resulting logic provides the tools necessary to prove a large class of example problems in a simple and efficient way.

[2]Thanks to Jacob Kornerup for first suggesting the connection between *co* and the notion of a timeout.

3 The Unity theory

The Unity theory was originally presented by Chandy and Misra in [CM88]. Recently a revised version has been proposed by Misra [Mis93]. In this section we present a slightly modified version of that in [Mis93].

3.1 Programs

A Unity program consists of four sections:

- A **declare** section. In this section the program variables and constants are declared.
- An **always** section. In this section certain invariants are asserted. This is useful for auxiliary variables.
- An **initially** section. This section contains state predicates which are asserted to be true for the initial state of the program.
- An **assign** section. This section contains a finite set of statements each of which is deterministic, terminating and has a well defined *wp* semantics. Frequently we will write $s \in F$ to indicate that statement s is in the **assign** section of program F. The *skip* statement is implicitly included in all programs.

An informal operational semantics for Unity programs is as follows. A computation begins in a state satisfying the **initially** section and loops forever, in each iteration non-deterministically selecting a statement from the **assign** section and executing it. The computation is required to be *fair* in the sense that each statement is selected infinitely often.

3.2 Properties

Ultimately, programs are created in order to produce computations. These computations are required to satisfy some set of properties known as the *specification* for the program. We introduce the following notation to indicate the claim, in theory T, that all computations satisfying A also satisfy B.

(3) $A \vdash_T B$

Usually, we will use this notation with a program F and a property P, as in $F \vdash_T P$, to indicate that all computations of F satisfy P. Note however that it makes sense as well to write $P \vdash_T Q$ for properties P and Q. Frequently we will drop the reference to the particular theory when it is known from the context. In the rest of this section we will only be concerned with the Unity proof system so we will write $F \vdash P$ to indicate that there is a proof of property P for program F in the Unity theory.

Program F defines a number of variables and constants, and the cross product of their ranges defines a state space for F. Following Dijkstra and Scholten [DS90], we will write $[p]_F$ to indicate (the static property) that predicate p holds everywhere in the state space of program F. The dynamic properties of F all arise from its static properties.

Also note that we will abuse notation somewhat by using the usual propositional connectives both in the object language and in the meta-language. For instance we state the transitivity of $\mapsto$ as

$$(F \vdash p \mapsto q) \wedge (F \vdash q \mapsto r) \Rightarrow (F \vdash p \mapsto r)$$

The above formula states that from a proof that $p \mapsto q$ is a property of F and a proof that $q \mapsto r$ is a property of F we can construct (by applying this rule) a proof that $p \mapsto r$ is a property of F. With the proper use of parentheses no confusion should arise. The benefit is that we can obtain formal proofs of some meta-theorems.

3.2.1 Safety

A safety property essentially states that *something bad does not happen*. A fundamental notion of safety in Unity is *invariance*. An invariant predicate is one which is always true. Hence any predicate that is true everywhere in the state space of F is an invariant of F. We indicate invariance of p as $\Box p$.

(4) $\quad [p]_F \Rightarrow (F \vdash \Box p)$

In Unity the fundamental notion of safety is *co* (short for *constrains*). A *co* property can be introduced by the following rule.

(5) $\quad (\forall s : s \in F : F \vdash \Box(p \Rightarrow wp.s.q)) \Rightarrow (F \vdash p \ co \ q)$

This rule in conjunction with (4) allows us to derive the following rule, which is frequently used to introduce a *co* property.

(6) $\quad (\forall s : s \in F : [p \Rightarrow wp.s.q]_F) \Rightarrow (F \vdash p \ co \ q)$

Informally, the meaning of $p \ co \ q$ is that whenever p holds, q also holds and q will hold in the next state. Note that since the *skip* statement is implicitly included in all programs, the left side of (5) implies $\Box(p \Rightarrow q)$. This is important for stuttering closure, i.e. so that the addition of a stuttering step to a computation which satisfies a *co* property does not invalidate the *co* property. The old notion of safety from [CM88], *unless*, can be stated in terms of *co*.

$$(F \vdash p \ unless \ q) \quad \equiv \quad (F \vdash p \wedge \neg q \ co \ p \vee q).$$

Another important notion of safety is that of stability. A stable predicate is one which, once it becomes true, remains true forever. Stability is just a special instance of *co*.

(7) $(F \vdash p \;\; st) \;\equiv\; (F \vdash p \;\; co \;\; p)$

Another way of proving $\Box p$ is by showing that p is true initially and that p is stable.

(8) $[F.initially \Rightarrow p]_F \wedge (F \vdash p \;\; st) \Rightarrow (F \vdash \Box p)$

3.2.2 Progress

The empty program, that with *skip* as its only statement, is completely safe. Hence safety properties are not useful by themselves. We need a way of asserting that *something good does happen.* In Unity this is done with $\mapsto$ (pronounced "leads-to"). Intuitively, $p \mapsto q$ means that whenever p becomes true, then either q is also true or q will become true at some time in the future.

The $\mapsto$ operator is based on the notion of transience. We write $p \;\; tr$ to indicate that p is transient, i.e. that whenever p becomes true, it will not remain true forever. A transient property is introduced by (9). Due to the fairness assumption, the existence of a single statement that will falsify a predicate is sufficient to guarantee the transience of that predicate.

(9) $(\exists s : s \in F : F \vdash \Box(p \Rightarrow wp.s.\neg p)) \Rightarrow (F \vdash p \;\; tr)$

From *tr* and *co* we construct $\mapsto$. Rule (10) is essentially the old *ensures* rule from [CM88]. The operator $\mapsto$ is the transitive (11) and disjunctive (12) closure of *ensures*.

(10) $(F \vdash p \wedge \neg q \;\; co \;\; p \vee q) \wedge (F \vdash p \wedge \neg q \;\; tr) \Rightarrow (F \vdash p \mapsto q)$

(11) $(F \vdash p \mapsto q) \wedge (F \vdash q \mapsto r) \Rightarrow (F \vdash p \mapsto r)$

(12) $(\forall p : p \in S : F \vdash p \mapsto q) \Rightarrow (F \vdash (\exists p : p \in S : p) \mapsto q)$

3.2.3 Derived rules

In large part the power of Unity comes from its *derived rules.* These rules can be proved from the fundamental axioms and inference rules already given. We present just a few of them here. The reader interested in others should consult [CM88] or [Mis93].

(13) $(F \vdash \Box(p \Rightarrow q)) \Rightarrow (F \vdash p \mapsto q)$

(14) $(F \vdash p \mapsto q) \Rightarrow (F \vdash p' \wedge p \mapsto q)$

(15) $(F \vdash p \mapsto q) \Rightarrow (F \vdash p \mapsto q \vee q')$

(16) $(F \vdash p \mapsto q \vee r) \wedge (F \vdash r \mapsto s) \Rightarrow (F \vdash p \mapsto q \vee s)$

(17) $(\forall m :: F \vdash p \wedge M = m \mapsto (p \wedge M < m) \vee q) \Rightarrow (F \vdash p \mapsto q)$
for m and state function M ranging over a set with well founded relation $<$.

(18) $(F \vdash p \mapsto q) \wedge (F \vdash r \ co\ s) \Rightarrow (F \vdash p \wedge r \mapsto (q \wedge s) \vee (\neg r \wedge s))$

Theorem (18) is known as the *PSP rule*. The name *PSP* stands for *progress, safety, progress* and comes from the fact that it enables one to derive a progress property from a progress property and a safety property. The PSP rule is very important and useful in the Unity theory.

4 The Real-Time Unity theory

Fundamental to the notion of real-time systems is the assumption that the programmer has control to some extent over the timing of certain statements. That is, they can guarantee under certain conditions that the statement will fire or that it will not fire within a certain time period. Hence the logic of Real-Time Unity includes two new constructs: *new.p* and *fires.s.k* where s is a statement, p is a predicate, and k is a time value. In this section we introduce these constructs and, based on them we define the bounded versions of the usual Unity operators.

4.1 Real-Time Unity Programs

Real-Time Unity programs are Unity programs with two extensions: statements are labeled, and assertions can be added to control the timing of the statements. All computations of the program are assumed to meet these timing constraints.

The timing assertions are of the form $\Box(X \Rightarrow \mathit{fires.s.k})$ or $\Box(X \Rightarrow \neg \mathit{fires.s.k})$, where k is a time value, s is a statement label, and X is either p or *new.p* for some state predicate p. We leave unspecified at this point the type of k. The intuitive meanings of these new predicates are given below.

(19) *new.p* holds at those points in a computation where p holds and p did not hold in the previous step. If p holds initially then *new.p* holds initially.

(20) *fires.s.k* holds at those points in a computation where statement s fires within k time units.

We assume that the statements of a Real-Time Unity program are terminating, deterministic and have a well defined *wp* semantics. As in standard Unity, a special statement labeled *skip* is implicitly included in all programs.

4.2 The Real-Time Unity logic

We now present the axioms and inference rules which comprise the Real-Time Unity logic. In section 4.3 we axiomatize *new* and *fires*. In section 4.4 we axiomatize the safety properties co_k and st_k. In section 4.5 we axiomatize the progress properties tr_k and $\mapsto_k$.

4.3 Axioms for *new* and *fires*

We axiomatize *new* and *fires* with the following axioms. We use the notation $F \ \vdash \ P$ to indicate that P is a property of the RTU program F. In the following, F is any Real-Time Unity program and s is any statement in F.

(21) $F \ \vdash \ \Box(\mathit{fires}.s.\infty)$

(22) $j < k \Rightarrow (F \ \vdash \ \Box(\mathit{fires}.s.j \Rightarrow \mathit{fires}.s.k))$

(23) $F \ \vdash \ \Box(\mathit{new}.q \Rightarrow q)$

(24) $F \ \vdash \ \neg q \ \ co \ \ \neg q \vee \mathit{new}.q$

(25) $(F \ \vdash \ \neg p \wedge q \ \ co \ \ \neg p \vee \neg q) \Rightarrow (F \ \vdash \ \Box(\mathit{new}.(p \wedge q) \Rightarrow \mathit{new}.p))$

4.4 Safety properties

We generalize the Unity operator co by adding a duration k as a subscript. Whereas the unbounded form $p \ \ co \ \ q$ applies to an entire computation, the property $p \ \ co_k \ \ q$ applies only to intervals of duration k which begin when p becomes true. The meaning of $p \ \ co_k \ \ q$ can be expressed as follows: starting at a point where p becomes true, if p fails within time k, then q holds as long as p does and at the point at which p fails. If p holds for time k then q must as well. After time k there is no requirement on p and q.

(26) $(\forall s :: (F \ \vdash \ \Box(\mathit{new}.p \Rightarrow \neg \mathit{fires}.s.k)) \vee (F \ \vdash \ \Box(p \Rightarrow \mathit{wp}.s.q)))$
$\Rightarrow \ \ (F \ \vdash \ p \ \ co_k \ \ q)$

(27) $(F \ \vdash \ p \ st_k) \ \ \equiv \ \ (F \ \vdash \ p \ \ co_k \ \ p)$

4.5 Progress properties

In this section we develop a notion of quantitative progress by generalizing the traditional Unity operators. Thus instead of properties of the form $p \ \ tr$ and $p \ \mapsto \ q$ we have properties such as $p \ \ tr_k$ and $p \ \mapsto_k \ q$. As in traditional Unity we begin by defining transience, tr_k (28). We then define $p \ \mapsto_k \ q$ inductively in (29),(30) and (31). Note that these are directly analogous to the rules for Unity $\mapsto$ (9),(10),(11) and (12).

(28) $(\exists s :: F \vdash \Box(new.p \Rightarrow fires.s.k) \wedge \Box(p \Rightarrow wp.s.\neg p)) \;\Rightarrow\; p \; tr_k$

(29) $(F \vdash p \wedge \neg q \; co_k \; p \vee q) \wedge (F \vdash p \wedge \neg q \; tr_k) \Rightarrow (F \vdash p \mapsto_k q)$

(30) $(F \vdash p \mapsto_j q) \wedge (F \vdash q \mapsto_k r) \Rightarrow (F \vdash p \mapsto_{j+k} r)$

(31) $(\forall p : p \in S : F \vdash p \mapsto_k q) \Rightarrow (F \vdash (\exists p : p \in S : p) \mapsto_k q)$

4.6 Derived rules

We also have a number of derived rules.

(32) $(F \vdash p \; co_\infty \; q) \;\equiv\; (F \vdash p \; co \; q)$

(33) $j < k \;\wedge\; (F \vdash p \; tr_j) \;\Rightarrow\; (F \vdash p \; tr_k)$

(34) $(F \vdash q \; tr_k) \;\wedge\; (F \vdash \Box(p \Rightarrow q)) \;\Rightarrow\; (F \vdash p \; tr_k)$

(35) $(F \vdash p \; tr_\infty) \;\equiv\; (F \vdash p \; tr)$

(36) $(F \vdash p \mapsto_\infty q) \;\equiv\; (F \vdash p \mapsto q)$

(37) $(F \vdash p \mapsto_j q) \;\wedge\; j < k \;\Rightarrow\; (F \vdash p \mapsto_k q)$

(38) $(F \vdash \Box(p \Rightarrow q)) \Rightarrow (F \vdash p \mapsto_k q)$

(39) $(F \vdash p \mapsto_j q \vee r) \wedge (F \vdash r \mapsto_k s) \Rightarrow (F \vdash p \mapsto_{j+k} q \vee s)$

(40) $(F \vdash p \mapsto_k \mathit{false}) \Rightarrow (F \vdash \Box\neg p)$

(41) $(\forall i : i \in \mathbb{N} : F \vdash p \wedge m = i \mapsto_{k.i} (p \wedge m < i) \vee q)$
$\Rightarrow (F \vdash p \wedge m = n \mapsto_K q)$
where $K = (\Sigma i : 0 \leq i \leq n : k.i)$.

From theorems (32),(35) and (36) we see that the axioms for *co*, *tr* and $\mapsto$ specialize to the usual axioms for Unity. The only additional axioms so far are (21), (22), (23),(24) and (25).

4.7 Combining safety and progress

In Unity, one of the most powerful derived rules is the *PSP* rule (18). The need to put progress and safety together to yield a new progress property occurs quite often. In Real-Time Unity we can expect this situation to be even more pronounced. In a fundamental way real-time progress and safety properties seem to arise in pairs. There is no reason for one process to go quickly unless another is going slowly. Conversely, there is no reason for one process to go slowly unless another is going quickly. Hence in the development of a theory of real-time computation, we should pay considerable attention to the interplay between bounded safety and progress. Below we present two combination rules which are very useful.

(42) $(F \vdash p \mapsto_k q) \wedge (F \vdash r \ co_k \ s)$
$\quad \Rightarrow (F \vdash p \wedge new.r \mapsto_k (q \wedge s) \vee (\neg r \wedge s))$

(43) $(F \vdash p \ st_k) \wedge (F \vdash q \ tr_k) \wedge (F \vdash p \wedge \neg q \ co \ \neg q)$
$\quad \Rightarrow (F \vdash p \wedge q \ co \ p)$

Theorem (42) is the real-time *PSP* theorem. Theorem (43) is called the *PSS* theorem (for *progress, safety, safety*). These theorems will be used in section 5 to prove some of the example problems.

5 Applications

5.1 Send/Acknowledge

As our first example we give a simple application of the cancellation rule (39). A sender sends a message to a receiver and awaits a response. It is known that within time j the message is either lost or received. It is also known that if it is received, it is acknowledged within time k. The goal is to show that the message is either lost or acknowledged within time $j + k$.

1. $sent \mapsto_j lost \vee recd$	; assume
2. $recd \mapsto_k ack$	; assume
3. $sent \mapsto_{j+k} lost \vee ack$	; (39)

5.2 Fischer's Protocol

In this section we present and prove Fischer's algorithm for mutual exclusion. The algorithm was first presented in [Lam87] and proofs of its correctness appear in [AL92] and [SBM92]. In addition to the proof in this section, in section 6 we show how the protocol can be derived from an untimed Unity program.

The algorithm uses timing constraints to ensure mutually exclusive access to a critical section. In the program, process index i ranges from 1 to n. Each process has a local state variable s which ranges over four states: a, b, c, d. We will sometimes abbreviate $s_i = a$ by a_i, etc. The processes also share a variable x, which can be viewed as a token which either takes the value of some process i in which case it is owned by process i or the value 0 in which case it is owned by no process. The algorithm may be intuitively appreciated by noting that once process i goes to state c, thereby setting x to i, no other process can enter state b. Process i then waits in state c for time k, thus ensuring that any process in state b has left state b. The last process to go from b to c will get to continue to the critical section d.

```
**Program** Fischer
**initially**
    $(\forall i :: s_i = a)$
**assign**
    $( [\!] \; i ::$
        $\alpha_i : s_i := b$                $if \;\; s_i, x = a, 0$
      $[\!] \; \beta_i : s_i, x := c, i$          $if \;\; s_i = b$
      $[\!] \; \gamma_i : s_i := d$               $if \;\; s_i, x = c, i$
      $[\!] \; \delta_i : s_i, x := a, 0$         $if \;\; s_i, x = a, 0$
    $)$
**assert**
    $(\forall i ::$
     $\Box(new.b_i \Rightarrow fires.\beta_i.k)$
     $\Box(new.c_i \Rightarrow \neg fires.\gamma_i.k)$
    $)$
**end**
```

We now prove Fischer's algorithm using three traditional Unity operators, *co*, *st* and $\Box$, and two of the new operators, st_k and tr_k. The crux of the proof is in step 11 where (43) is applied. The rest of the proof is standard Unity. All of the steps labeled "from text" have been verified by a Unity model checker developed by Markus Kaltenbach [Kal93].

(44) $$\Box(d_i \Rightarrow x_i)$$

Proof:

1. $a_i \;\; co \;\; a_i \vee b_i$	; from text
2. $b_i \;\; co \;\; b_i \vee (c_i \wedge x_i)$	; from text
3. $c_i \wedge \neg x_i \;\; co \;\; c_i \wedge \neg x_i$	; from text
4. $d_i \wedge x_i \wedge \neg(\exists j :: b_j) \;\; co \;\; (d_i \wedge x_i \wedge \neg(\exists j :: b_j)) \vee a_i$	; from text
5. $c_i \wedge x_i \wedge \neg b_j \;\; co \;\; \neg b_j$	; from text
6. $(c_i \wedge \neg x_i) \vee (c_i \wedge x_i \wedge \neg b_j) \;\; co \;\; \neg x_i \vee \neg b_j$	; 3, rhw, 5, disj.
7. $c_i \wedge \neg(x_i \wedge b_j) \;\; co \;\; \neg(x_i \wedge b_j)$	; 6 pred. calc.
8. $c_i \;\; st_k$	; (26),(27)
9. $b_j \;\; tr_k$	; (28)
10. $x_i \wedge b_j \;\; tr_k$	; 9, (34)
11. $c_i \wedge x_i \wedge b_j \;\; co \;\; c_i$	; 7,8,10, (43)
12. $c_i \wedge x_i \wedge (\exists j :: b_j) \;\; co \;\; c_i$	; 11 disj.
13. $(c_i \wedge x_i \wedge \neg(\exists j :: b_j) \;\; co \;\; (d_i \wedge x_i \wedge \neg(\exists j :: b_j))$	; from text
14. $a_i \vee b_i \vee c_i \vee (d_i \wedge x_i \wedge \neg(\exists j :: b_j)) \;\; st$	; 1,2,3,4,12,13 disj.
15. $initially \; a_i$	; assume
16. $\Box(d_i \Rightarrow x_i)$	; 14,15, pred. calc.

5.3 Timeout

In this section we relate the formal notion of quantitative safety in the form of co_k to the waiting that a process does in a timeout protocol. Process α tries to establish a connection with β which may be up, in which case β will acknowledge the attempt within time k, or $\neg up$, in which case β will simply fail to respond. Process α will timeout after waiting for time k. The state of the system is represented by the variable $x \in \{t, a, f\}$, representing *trying*, *acknowledged* and *failed.*

```
Program timeout
assign
    $\alpha :\ \ x := f \ \ if \ \ x = t$
    $\beta :\ \ x := a \ \ if \ \ x = t \wedge up$
assert
    $\Box(new.(x = t \wedge up) \Rightarrow \neg fires.\alpha.k)$
    $\Box(new.(x = t \wedge up) \Rightarrow fires.\beta.k)$
end.
```

We show that an attempt by α to establish a connection succeeds if β is up and fails if β is not up. We do this by proving the following two theorems.

(45) $t \wedge up \ \mapsto \ a$

Proof:

1. $t \wedge up \wedge \neg a \ \ tr_k$	; β, (28)
2. $t \wedge up \wedge \neg a \ \ co_k \ \ (t \wedge up) \vee a$	; α, (26)
3. $t \wedge up \ \mapsto_k \ a$	; 1,2, (29)

(46) $t \wedge \neg up \ \mapsto \ f$

Proof:

1. $t \wedge \neg up \wedge \neg f \ \ tr$	; α, (28)
2. $t \wedge \neg up \wedge \neg f \ \ co \ \ (t \wedge \neg up) \vee f$	; (26)
3. $t \wedge \neg up \ \mapsto \ f$	; 1,2, (29)

Note especially line 2 of the first proof. This is the sense in which co_k captures the notion of a timeout. Once α is trying and is not acknowledged, it continues to try for time k before giving up.

6 Weakening the Guard

We introduce a refinement rule for real time programs which we call *weakening the guard.* This allows us to develop a real time Unity program from an untimed Unity program, thereby simplifying the derivation process. We then show how the rule may be applied to three algorithms: the timeout algorithm and Fischer's mutual exclusion protocol from section 5, and a real-time producer/consumer single element buffer algorithm.

The idea is to refine a program by replacing one of its statements with another statement with a weaker guard. The theorem says that under certain conditions we can do this, impose some appropriate timing constraints and preserve all the properties of the original program, modulo substitution of the new statement label for the old one.

(47) **Theorem:** Let F be a RTU program, α and α' statements, P a set of predicates, q a predicate and k a time value such that

(47.0) $[grd.\alpha \Rightarrow grd.\alpha']$
(47.1) $[grd.\alpha' \Rightarrow (\exists p : p \in P : p)]$
(47.2) $(\exists p : p \in P : p) \;\mapsto_k\; q$
(47.3) $[q \Rightarrow (\forall r :: wp.\alpha'.r \;\equiv\; wp.\alpha.r)]$
(47.4) $(\forall s :: [\neg q \vee wp.s.q \vee (\exists p : p \in P : \neg p \wedge wp.s.p)])$
(47.5) $(\forall p : p \in P : \Box(new.p \Rightarrow \neg fires.\alpha'.k))$

then $(F \,[\!]\, \alpha \;\vdash\; X) \Rightarrow (F \,[\!]\, \alpha' \;\vdash\; (\alpha := \alpha').X)$
for any real time Unity property X.

6.1 Timeout Revisited

We now apply theorem (47) to the timeout problem. In the problem as presented in section 5 the purpose of having statement α wait is so that β can respond if it is up. Since β will fire in time k when it is up, α can assume that the system is not up if $x = t$ after waiting time k. A simpler way to model this is with the untimed program below. In *timeout0*, no timing constraints are imposed. Instead, statement α has the stronger guard $x = t \wedge \neg up$. The proof of the theorems is straightforward.

```
Program timeout0
assign
    $\alpha$ :  $x := f \;\; if \;\; x = t \wedge \neg up$
    $\beta$ :  $x := a \;\; if \;\; x = t \wedge up$
end.
```

Theorem: $t \wedge up \;\mapsto\; a$
Theorem: $t \wedge \neg up \;\mapsto\; f$

Now we can apply the refinement rule. We first require that α fires slowly and β fires quickly.

```
Program timeout1
assign
    $\alpha'$ :  $x := f$  if  $x = t$
    $\beta$ :  $x := a$  if  $x = t \wedge up$
assert
    $\Box(new.(x = t) \Rightarrow \neg fires.\alpha'.k)$
    $\Box(new.(x = t) \Rightarrow fires.\beta.k)$
end.
```

Theorem: *timeout1* refines *timeout0*
Proof sketch: We replace α with α'. Rule (47) can be applied with the following instantiations:

$[grd.\alpha \;\equiv\; x = t \wedge \neg up]$
$[grd.\alpha' \;\equiv\; x = t]$
$P \;\equiv\; \{x = t\}$
$q \;\equiv\; x = t \Rightarrow \neg up$
k is any time value.

6.2 Fischer Revisited

The weakening the guard rule can easily be applied to Fischer's algorithm. In Fischer the point of the γ statements waiting is so that all processes have time to leave state b. At this point the one which possesses the token x may proceed to its critical section. We can model this in untimed Unity by adding the additional conjunct $(\forall i :: \neg b_i)$ to the guard of statement γ. In the resulting untimed Unity program the invariant $\Box(d_i \Rightarrow x_i)$ is easy to show.

```
Program Fischer0
initially
    $(\forall i :: s_i = a)$
assign
    ( $[\!]$ $i$ ::
        $\alpha_i$ :   $s_i := b$          if  $s_i, x = a, 0$
      $[\!]$ $\beta_i$ :   $s_i, x := c, i$    if  $s_i = b$
      $[\!]$ $\gamma_i$ :   $s_i := d$          if  $s_i, x = c, i \wedge (\forall i :: \neg b_i)$
      $[\!]$ $\delta_i$ :   $s_i, x := a, 0$    if  $s_i = d$
    )
end.
```

Theorem: $\Box(d_i \Rightarrow x_i \wedge (\forall i :: \neg b_i))$

Proof: Straightforward Unity invariance proof.

Now we restate the timed version of Fischer and claim that rule (47) can be applied with the instantiations following the program.

```
Program Fischer1
initially
    $(\forall i :: s_i = a)$
assign
    $( [\!] \; i ::$
        $\alpha_i$:   $s_i := b$          $\mathit{if}\;\; s_i, x = a, 0$
      $[\!]$ $\beta_i$:   $s_i, x := c, i$    $\mathit{if}\;\; s_i = b$
      $[\!]$ $\gamma'_i$:   $s_i := d$          $\mathit{if}\;\; s_i, x = c, i$
      $[\!]$ $\delta_i$:   $s_i, x := a, 0$    $\mathit{if}\;\; s_i = d$
    $)$
assert
    $(\forall i ::$
     $\Box(new.b_i \Rightarrow \mathit{fires}.\beta_i.k)$
     $\Box(new.c_i \Rightarrow \neg \mathit{fires}.\gamma'_i.k)$
    $)$
end.
```

Theorem: Fischer1 refines Fischer0
Proof sketch: We replace γ with γ'.

$[grd.\gamma \;\; \equiv \;\; s_i, x = c, i \wedge (\forall i :: \neg b_i)]$
$[grd.\gamma' \;\; \equiv \;\; s_i, x = c, i]$
$P \;\; \equiv \;\; \{c_i\}$
$q \;\; \equiv \;\; x = i \Rightarrow (\forall i :: \neg b_i)$
k is any time value.

6.3 Single Element Buffer

In this section we present a simplified communications example in which timing constraints are used to introduce partial synchrony into an otherwise asynchronous program. The approach is to first present a fully synchronous approach and then relax the synchrony by introducing timing constraints.

In this example, a sender and a receiver communicate via a channel c of length 1. The sender σ writes to the channel c and then waits until the channel is empty before writing again. In the untimed version this waiting is modeled by the conjunct $c = [\,]$ in the guard of σ. The real-time version *SEB1* is refined from *SEB0* using (47).

```
Program SEB0
assign
    σ :  $c, in := [hd.in],\ tl.in$        $if\ in \neq [\,] \wedge c = [\,]$
    ρ :  $out, c := out; hd.c,\ [\,]$      $if\ c \neq [\,]$
end.
```

```
Program SEB1
assign
    σ :  $c, in := [hd.in],\ tl.in$        $if\ in \neq [\,]$
    ρ :  $out, c := out; hd.c,\ [\,]$      $if\ c \neq [\,]$
assert
    $\Box(new.(in = (x : xs)) \Rightarrow \neg fires.\sigma.k)$
    $\Box(new.(c \neq [\,]) \Rightarrow fires.\rho.k)$
end.
```

Theorem: SEB1 refines SEB0.
Proof sketch: We replace σ with σ'.

$[grd.\sigma \;\equiv\; in \neq [\,] \wedge c = [\,]]$
$[grd.\sigma' \;\equiv\; in \neq [\,]]$
$p \in P \;\equiv\; (\exists x, xs :: [p \;\equiv\; in = (x : xs)])$
$q \;\equiv\; c = [\,]$
k is any time value.

7 Related Work

Most, if not all real-time theories involve extensions to some untimed model of computation. Hence we see real-time automata [AD92, LV92], real-time process algebras [BB91, Dav93], interval logics [CHR91], real-time temporal logics [AL92, Eme92, Hen90, HMP92, Koy90], and special formalisms for real-time [Mok91, SBM92]. The differences between the Real-Time Unity approach and those approaches which extend a non-temporal logic model of computation are similar to the differences between their untimed counterparts. We will concentrate in this section on the different temporal logic approaches to real-time.

7.1 A taxonomy of real-time temporal logics

In [AH92] Alur and Henzinger classify real-time temporal logics along four semantic axes: *state sequences* or *observation sequences*, *time intervals* or *time points*, *strictly monotonic* or *weakly monotonic time* and *real-numbered time* or *integer time*. These semantic classifications are valid for all approaches to real-time. In addition, real-time temporal logics can be characterized by whether they are propositional or first-order, linear or branching-time, by which temporal operators are used, and by three ways of introducing timing constraints into the syntax: *bounded temporal operators*, *freeze quantification* and *explicit clock variable*.

Real-Time Unity, in its current incarnation, is a first-order, linear-time logic which uses bounded versions of the usual Unity operators to express timing constraints. In this paper we have deliberately avoided the decision as to whether the underlying model is discrete or dense and whether it is strictly or weakly monotonic. The only model for which Real-Time Unity is known to be sound is a dense time, strictly monotonic model. These choices are the result of certain design decisions in the development of the logic. We believe that the Real-Time Unity logic might be easily "ported" to another semantics, e.g. a weakly monotonic, integer time semantics. There is reason to suspect that the propositional fragment might be useful for a model checking approach but this has not yet been investigated.

7.2 Timed Transition Systems

Perhaps the closest relative to Real-Time Unity are the *Timed Transition Systems* of Henzinger, Manna and Pnueli [HMP91, HMP92]. There are some notable differences, however. Firstly, our timing constraints are not tied to the guard of the statement, whereas theirs are explicitly defined in terms of the transition being enabled. This allows us flexibility in the manipulation of properties since we can alter the trigger predicate without changing the guard of the statement. Secondly, we provide a deductive system at a high level of abstraction which includes rules for composing real-time progress and safety properties and a method of program refinement (section 6) which can be used to introduce timing constraints. The deductive system put forth in [HMP91] does not provide this ability which we feel is an important aspect of the development of real-time systems.

7.3 Explicit clock variable

In [AL92], Abadi and Lamport take the explicit clock variable approach. Their real valued clock variable, which they call *now*, is postulated to increase without bound and never to decrease. Other than that, *now* is just an ordinary variable of the logic. A number of *timers* are defined: *MinTime, MaxTime, PTimer* and *VTimer*. With these constructs they can write what are essentially Unity programs and more. Despite its lack of generality, we prefer the simplicity of Real-Time Unity.

It is also interesting to note a basic difference in the way what we call real-time progress properties are handled in [AL92]. Progress properties are written as safety properties of the form "*now* will not advance beyond k unless something happens". Of course this only works if *now* will advance without bound. Unless it is explicitly required to advance we have a problem known as *Zeno's paradox*. Hence they include a *non-Zeno* axiom.

In our formulation however, the progress properties stand on their own and require no such behavior of time. Time advancing in a Zeno manner does not cause any problem for our progress properties. This is particularly important for properties of the form $p \mapsto_{\infty} q$. If real time progress properties are formulated as safety properties, we lose the connection between $p \mapsto q$ and $p \mapsto_{\infty} q$, even with no Zeno behavior. Since time never reaches ∞ the latter property (in the progress as safety formulation) would not require the eventual establishment of q. This leads us not to use the progress as safety formulation. Having done this we do not require non Zeno behavior. (Of course a non-Zeno axiom is consistent with our approach so if it is needed for other reasons we should feel free to postulate it.)

7.4 General bounded operators

In [Koy90] Koymans extends traditional temporal or *tense* logic using a bounded operator approach. The semantics is based on the difference between points in the computation, and the logic includes bounded modal operators which allow one to express that a predicate will hold at a point exactly distance δ in the future. Universal and existential quantification allow the expression of "at some point in the future" and "at all points in the future". Moreover, since the semantics is a partial order or *true concurrency* semantics, one can express both that p will hold at *all* points distance δ in the future and that p will hold at *some* point distance δ in the future. Since they generalize the original tense logic, they also have operators which look to the past rather than the future. Koymans describes timeout as one of the easiest notions to express and indeed in their presentation a "timeout" is expressed as $\neg P_{<\delta}e$ which means that event e has not occurred in the past δ time units. With respect to the timeout example in section 5, it should be noted that what Koymans has presented here is just a timeout condition whereas we tried to present the notion of timeout in a larger context of correctness and actually prove the program. Koymans does not show how one might use his formulation to prove correctness of a program which uses a timeout. Indeed, no deduction system is provided at all. However, Metric Temporal Logic seems to be a fully general real-time logic and appears to this student as perhaps the best grounded semantically.

7.5 The freeze quantifier

In [Hen90] Henzinger proposes the *freeze* quantifier. In this approach no explicit time variable is used but local variables occur in expressions to capture the notion of time at a given point in the computation. Hence one can write the bounded response requirement as

(48) $\Box x.(p \Rightarrow \Diamond y.(q \wedge y \leq x + k))$

In this formulation, the "$x.$" *freezes* the value of x to the time at which (this instance of) p is true. Similarly the value of y is frozen to the time at which q becomes true. Hence (48) is essentially equivalent to the explicit time variable formula (49). (Of course the variable y is not really necessary here.)

(49) $\Box(p \wedge x = now \Rightarrow \Diamond(q \wedge y = now \wedge y \leq x + k))$

Hence we can see that the freeze quantifier notation allows one to replace "$\wedge\ x = now$" with just "$x.$", resulting in considerable syntactic economy while retaining a good deal of the expressibility with respect to the explicit time variable approach.

8 Future Work

In this section we propose areas for future research. The section is divided into four subsections: extensions to the logic, applications of the logic, semantics and mechanization.

8.1 Extensions to the Real-Time Unity logic

There are at least two ways that Real-Time Unity might be extended. One is by introducing a notion of composition of programs. The other is by expanding the role of refinement. We discuss these below.

8.1.1 Composition

The problem of composition of programs is an important one in any programming language or methodology. It has been the object of much study in the Unity world as well. The original *union rules* in [CM88] are not consistent with the substitution axiom. Hence other approaches have been proposed by Misra [Mis93] and others.

We are currently investigating another possibility for the composition of Unity programs which is based on the inclusion of a **safety** section in the program syntax. This section contains safety properties which restrict the computation set of this program as well as that of any program with which it is composed. The methodological upshot is that a programmer is forced to do some bookkeeping of the safety properties which are used in any progress proof. A second proof obligation is that all properties in the safety section are true in the sense that the corresponding Hoare triples hold in the entire (not just reachable) state space of the program. This approach is largely orthogonal to the issue of real time and should be equally applicable here.

8.1.2 Refinement Methodology

The *weakening the guard* refinement rule appears to be a powerful way to introduce timing constraints into a program. However, the *window of opportunity* example does not appear to be directly solvable in this manner. It is possible to prove this example with the use of auxiliary variables, however the derivation is rather unintuitive. We hope to develop other refinement rules that can be used for this problem and any others that seem important. A central idea in our work has been that progress and safety properties arise in pairs. If this is the case, refinement rules which introduce the properties as pairs should be extremely useful.

8.2 Applications of Real-Time Unity

8.2.1 Communications protocols

In [BGM91], Brown, Gouda and Miller present a communications protocol which requires the use of waiting. The formalization of their solution does not model this idea explicitly. Rather, guards are used to check for the global condition. However, the efficiency of the solution relies on the fact that a process need not actually check for the global condition but can instead wait for the required time and be assured that the condition has become true. We are applying the weakening-the-guard refinement rule (47) to improve the exposition of these protocols.

8.2.2 Hybrid Systems

Hybrid systems are systems in which digital computing systems interact with continuous physical systems. Computing languages and logics for such systems need some way of expressing integration of real world continuous functions over time. The canonical example is the *gas burner* problem [Lam92, CHR91]. Ways of approaching the problem in Real-Time Unity include:

- **Clocks:** At regular periods a statement updates a program variable. If the period is accurate enough the program variable approximates a real world variable (which is some function of time e.g. the gas in the burner).
- **Meters:** A statement updates a program variable by "reading" a real world variable. The more frequently this is done the more accurate the program variable will be. For example the meter might actually measure the gas concentration in the burner.

Either way, it comes down to the approximation of a continuous variable by a discrete variable. Hence an invariant relation asserting the closeness of the approximation is central to such an approach.

8.3 Semantics for Real-Time Unity

We plan to investigate various semantic issues: discrete vs. dense time, interleaved vs. true concurrency and strictly vs. weakly monotonic. We will prove soundness in at least one variant. We will consider the problem of completeness but it is not a primary goal of ours. Consistent with the approach in [CM88] we plan instead to show that the theory is widely applicable.

8.4 Mechanization of Real-Time Unity

The Unity theory has been automated in HOL [And92] and in the Boyer-Moore computational logic [Gol92]. In addition, the implementation of a Unity based model checker is underway [Kal93]. We wish to consider extending one of these models to include Real-Time Unity. A particularly attractive approach would be to use the model checker as a decision procedure to be called from a theorem prover for Real-Time Unity.

References

[AD92] Rajeev Alur and David Dill. The theory of timed automata. In de Bakker et al. [dB+92].
AFN: 545.

[AH92] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In de Bakker et al. [dB+92].

[AL92] M. Abadi and L. Lamport. An old fashioned recipe for real-time. In de Bakker et al. [dB+92].

[Alu91] Rajeev Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Stanford University, 1991.

[And92] Flemming Andersen. A theorem prover for UNITY in Higher Order Logic. Technical report, TFL, Lyngsø Allé 2, DK-2970 Hørsholm, Denmark, March 1992.

[BB91] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[BGM91] Geoffrey M. Brown, Mohamed G. Gouda, and Raymond E. Miller. Block acknowledgment: Redesigning the window protocol. *IEEE Tansactions on Communications*, 39(4):524–532, April 1991.

[CHR91] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[CM88] Mani Chandy and Jayadev Misra. *Parallel Programming: A Foundation*. Addison-Wesley, 1988.

[Dav93] Jim Davies. *Specification and Proof in Real-Time CSP*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1993.

[dB+92] J. W. de Bakker et al., editors. *Real-Time: Theory in Practice: REX Workshop*, volume 600 of *LCNS*, New York, NY, July 1992. Springer-Verlag New York, Incorporated.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16. MIT Press, 1990.

[Eme92] E. Allen Emerson. Real-time and the mu-calculus. In de Bakker et al. [dB+92].

[Gol92] David Goldschlag. Mechanically verifying concurrent programs. Technical Report 71, Computational Logic Inc., Austin, TX, 1992.

[HC92] M.R. Hansen and Zhou Chaochen. Semantics and completeness of duration calculus. In de Bakker et al. [dB$^+$92].

[Hen90] Thomas A. Henzinger. Half-order modal logic: how to prove real-time properties. In *Proceedings of the Ninth Annual symposium on Principles of Distributed Computing*, pages 281–296. ACM, 1990.

[Hen91] Thomas A. Henzinger. *The Temporal Specification and Verification of Real-Time Systems*. PhD thesis, Stanford University, 1991.

[HMP91] T.A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for real-time systems. In *Proceedings of the 18th Annual symposium on Principles of Programming Languages*, pages 353–366. ACM Press, 1991.

[HMP92] T.A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In de Bakker et al. [dB$^+$92].

[Kal93] Markus Kaltenbach. The UNITY verifier, 1993. Markus is developing a model checker for Unity here at UT-Austin.

[Koy90] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2:255–299, 1990.

[Koy92] R. Koymans. (Real) time: A philosophical perspective. In de Bakker et al. [dB$^+$92].

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, Feb 1987.

[Lam91] Leslie Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Corporation, Systems Research Center, Dec 1991.

[Lam92] Leslie Lamport. Hybrid systems in TLA$^+$. In *Proceedings of a Workshop on Theory of Hybrid systems*, Lyngby, Denmark, October 1992.

[LV92] Nancy Lynch and Frits Vaandrager. Forward and backward simulations for timing-based systems. In de Bakker et al. [dB$^+$92].

[Mis93] J. Misra. untitled. This is an unpublished manuscript on a new version of the Unity logic, 1993.

[Mok91] Aloysius K. Mok. Towards mechanization of real-time. In van Tilborg and Koob [vTK91].

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Providence, 1977.

[SBM92] F.B Schneider, B. Bloom, and K. Marzullo. Putting time into proof outlines. In de Bakker et al. [dB+92].

[vTK91] André M. van Tilborg and Gary M. Koob, editors. *Foundations of Real-Time Computing - Formal Specifications and Methods.* Kluwer Academic Publishers, 1991.