TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

A compositional proof system for real-time systems based on explicit clock temporal logic

Citation for published version (APA):
Hooman, J. J. M., Kuiper, R., & Zhou, P. (1991). A compositional proof system for real-time systems based on explicit clock temporal logic. In *Proceedings Sixth International Workshop on Software Specification and Design (Como, Italy, October 25-26, 1991)* (pp. 110-117). Institute of Electrical and Electronics Engineers. https://doi.org/10.1109/IWSSD.1991.213070

DOI:
10.1109/IWSSD.1991.213070

Document status and date:
Published: 01/01/1991

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 30. Sep. 2022

A Compositional Proof System for Real-Time Systems Based on Explicit Clock Temporal Logic*

J. Hooman R. Kuiper P. Zhou

Dept. of Mathematics and Computing Science
Eindhoven University of Technology
P.O.Box 513
5600 MB Eindhoven, The Netherlands

Abstract

To specify timing properties of real-time systems, we consider Explicit Clock Temporal Logic. Programs are written in an Occam-like real-time language. A proof system is provided to formally verify that a program satisfies a specification expressed in our real-time version of temporal logic. The proof system is compositional, sound, and relatively complete.

1 Introduction

In this paper we investigate the formal specification and verification of real-time systems. One of our objectives was that the approach should be able to deal with a reasonably realistic language. Therefore, we consider an Occam-like real-time programming language [8] with synchronous message passing along unidirectional channels between concurrent processes. Another aim was to build on existing formalisms and enable easy future extension or adaptation. In general, to specify and verify real-time systems, an existing non-real-time method can be extended with some notion of time. Here we also follow this approach, and use an extension of linear time temporal logic [12, 10]. The standard logic enables the expression of safety properties (such as "non-termination", "no communication along channel c", and "program variable x is always greater than 5") as well as liveness properties (such as "termination", "eventually communicate along channel c", and "eventually x has the value 8"). To specify real-time properties (such as "termination within 7 time units", "communicate on channel c at time 4", and "during the next 3 time units x has a positive value") we have to extend temporal logic with a quantitative notion of time. As already observed in [4], one can distinguish two main approaches.

In one approach, new temporal operators are introduced by extending the standard ones with time bounds. A general discussion of this extension in the context of linear time, often called Metric Temporal Logic (MTL), can be found in [9]. In a similar way, branching time temporal logic, also called Computation Tree Logic (CTL), is extended to real-time by adding time bounds to the modal operators. See, for instance, [2] where also algorithms for model checking and satisfyability analysis are presented.

We investigate the alternative approach in which temporal logic is extended with a distinguished variable that explicitly refers to clock time. Such a logic has been used in [11] to reason about real-time discrete event systems. A similar real-time version can be found in [3]. This approach has the advantage of building upon a well-established classical temporal logic framework to which only axioms and rules for proving timing properties have to be added. Also, further extensions can be envisaged, like extra structure on the time domain, that leave the underlying logic unchanged. It is even possible to leave choices about the time domain, like this being discrete or dense, open till later. This allows for investigating the effect of different choices on, e.g., the possibilities for automatic verification without drastic changes to the formalism.

Given the choice of the programming language and the real-time version of temporal logic, the task is to develop a proof method to verify that a program satisfies a property expressed in this logic. In classical verification methods, such as [10] for temporal logic, the complete program text must be available. Global proof systems, based on the method of [10], and deci-

*This research was supported by ESPRIT-BRA project 3096 "Formal Methods and Tools for the Development of Distributed and Real-Time Systems (SPEC)".

TH0388-9/91/0000/0110$01.00 © 1991 IEEE

sion procedures for explicit clock versions of temporal logic can be found in [3, 11].

In contrast with these methods, we formulate a *compositional* proof system in which the specification of a compound programming language construct (such as sequential composition and parallel composition) can be deduced from specifications for its constituent parts without any further information about the internal structure of these parts. Compositionality can be considered as a prerequisite for hierarchical, structured, program derivation. By means of a compositional proof system the design steps can be verified during the process of top-down program construction.

To obtain a compositional proof system, our starting point is the compositional system for classical temporal logic as described in [1]. The present paper also builds on the approach to compositionality and real-time introduced in [7]. With respect to that paper, the main difference is that the emphasis there was on achieving compositionality in as simple a setting as possible. Therefore, a quite sparse model of computation was used. Furthermore, that paper forms a complement to the present one in the sense that the logic was MTL-styled rather than equipped with an extra time variable. Related compositional proof systems based on MTL for variations of the programming language have been formulated in [6].

This paper is structured as follows. In Section 2 we describe our programming language and the basic timing assumptions. An outline of the semantic model is given in Section 3. In Section 4 we define our version of explicit clock temporal logic. A compositional proof system is formulated in section 5. Finally, Section 6 contains concluding remarks. More technical details can be found in [13].

2 Real-Time Programming Language

2.1 Syntax and Informal Semantics

We consider a real-time programming language which is akin to Occam [8]. Concurrent processes communicate and by synchronous message passing via unidirectional channels which connect exactly two processes.

Let VAR be a nonempty set of program variables, $CHAN$ be a nonempty set of channel names, and VAL be the domain of values of program variables. In our real-time programming language we have the following statements, with $c, c_i \in CHAN$ and $x, x_i \in VAR$.

Atomic statements

- **skip** terminates immediately.
- $x := e$ assigns the value of expression e to x.
- **delay** e suspends execution for (the value of expression) e time units.
- $c!e$ sends the value of expression e on channel c as soon as a corresponding input statement is available. Since we assume synchronous communication, such an output statement is suspended until a parallel process executes an input statement of the form $c?x$.
- $c?x$ receives a value via channel c and assigns this value to the variable x. Similar to the output statement, such an input statement has to wait for a corresponding output statement before a synchronous communication takes place.

Compound statements

- $S_1; S_2$ indicates sequential composition.
- Guarded command $[[]_{i=1}^{n} b_i \rightarrow S_i]$ is executed as follows. If none of the b_i evaluates to true then this command terminates after the evaluation of the booleans. Otherwise, non-deterministically select one of the b_i's which evaluates to true and execute the corresponding statement S_i.
- Guarded command $[[]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i \,[]\, b; \textbf{delay } e \rightarrow S]$ is executed as follows. A guard $b_i; c_i?x_i$ or $b;$ **delay** e is *open* if the boolean part evaluates to true. If none of the guards is open, the guarded command terminates after evaluation of the booleans. Otherwise, wait until an io-statement of the open io-guards can be executed and continue with the corresponding S_i. If the delay guard is open (b evaluates to true) and no io-guard can be taken within e time units (after the evaluation of the booleans), then S is executed. Boolean expressions equivalent to true are often omitted in guards.

  Example 2.1 Observe that delay-values can be arbitrary expressions, for instance,
  $d?x; [d?y \rightarrow S_1 \;[]\, c?x \rightarrow S_2 \;[]\, \textbf{delay } (x+6) \rightarrow S_3]$

  Example 2.2 By means of a guarded command, we can easily express a time-out. For instance,
  $[x > 0; c?y \rightarrow x := y + x \;[]\; \textbf{delay } 10 \rightarrow \textbf{skip}]$
  informally means that if $x > 0$ and the input communication can take place within 10 time units then the assignment is executed, otherwise after 10 time units there is a time-out and the skip-statement is executed.
- $\star G$ indicates repeated execution of guarded command G as long as one of the guards is open. When none of the guards is open, $\star G$ terminates.
- $S_1 \| S_2$ indicates parallel execution of S_1 and S_2. No program variable should occur in both S_1 and S_2.

Henceforth we use $\equiv$ to denote syntactic equality.

Define $var(S)$ as the set of variables occurring in statement S. The set of (directional) channels occurring in S, denoted by $dch(S)$, is defined as the set containing all channels occurring in S together with all directional channels $c!$ and $c?$ occurring in S. For instance, $dch(c!5; d?y \| c?x) = \{c, c!, c?, d, d?\}$.

2.2 Basic Timing Assumptions

In order to describe the real-time behavior of programs, we must make assumptions about the execution time of atomic statements and the extra time needed to execute compound constructs. In our proof system, the correctness of a program with respect to a specification, which may express timing properties, is verified relative to these assumptions. For simplicity, we assume in this paper that a **delay** e statement takes exactly e time units. Furthermore we assume given positive constants K_a, K_c and K_g such that every assignment takes K_a time units, each communication takes K_c time units, and the evaluation of the guards in a guarded command takes K_g time units. There is no overhead for other compound statements.

The most important assumption involves parallel composition. Clearly, the execution time of a simple program $x := 0 \parallel y := 1$ depends on the number of available processors. In general, we have to make an assumption about the assignment of processes to processors at parallel composition. In this paper we use the *maximal parallelism* model to represent the situation that each process has its own processor. Consequently any action is executed as soon as possible. Observe that maximal parallelism implies *minimal waiting*: a process only waits when it tries to execute an input or output statement and the communication partner is not available. This maximal parallelism assumption has been generalized in [5] to multiprogramming where several processes can be executed on a single processor and scheduling is based on priorities which can be assigned to statements in the program.

3 Denotational Semantics

A good starting point for the development of a compositional proof system is the formulation of a denotational, and hence compositional, semantics. In such a semantics the meaning of a statement must be defined without any information about the environment in which it will be placed. Hence, the semantics of a statement in isolation must characterize all potential computations of the statement. When composing this statement with (part of) its environment, the semantic operators must remove the computations that are no longer possible. To be able to select the correct computations from the semantics, any dependency of an execution on the environment must be made explicit in the semantic model.

In our semantics the timing behaviour of a program is expressed from the viewpoint of an external observer with his own clock. Let this clock range over a time domain *TIME*. Thus, although parallel components of a system might have their own, physical, local clock, the observable behaviour of a system is described in terms of a single, conceptual, global clock. Since this global notion of time is not incorporated in the distributed system itself, it does not impose any synchronization upon processes.

To define the timing behaviour of a statement **delay** e, we have to relate expressions in the programming language to our time domain. For simplicity we have assumed that **delay** e takes e time units and hence, implicitly, that $VAL \subseteq TIME$. Furthermore we assume that the standard operators $+$, $-$, $\times$ and $\leq$ are defined on *TIME*. In examples we often assume that *VAL* includes the natural numbers. Note that we allow our time domain to be dense (a domain is dense if between every two points there exists a third point).

Henceforth, we use $\tau, \tau_0, \tau_1, \ldots$ to denote values from *TIME*. For notational convenience, we use a special value ∞ with the usual properties, such as $\infty \notin TIME$, and for all $\tau \in TIME \cup \{\infty\}$: $\tau \leq \infty$, $\tau + \infty = \infty + \tau = \infty$, etc.

A computation of a program is represented by a mapping which assigns to each point of time during this computation a pair consisting of a state and a set of communication records. The state represents the values of the program variables at that point of time.

Definition 3.1 (States) The set of states *STATE* is defined as the set of mappings from *VAR* to *VAL*, $STATE = \{ s \mid s : VAR \rightarrow VAL \}$.

Thus a state $s \in STATE$ assigns to each program variable x a value $s(x)$.
Communication records denote the state of affairs on the channels of the program. To denote the real-time communication behaviour, we use records of the form (c, ϑ), indicating a communication along channel c with value ϑ.

Definition 3.2 (Communication Records) $COMM = \{(c, \vartheta) \mid c \in CHAN \text{ and } \vartheta \in VAL \}$.

In addition to the communications at any point of time, the model includes information about those processes waiting to send or waiting to receive messages

on their channels at any given time. We use wait-records of the form $c!$ and $c?$ to indicate that a process is, respectively, waiting to send or waiting to receive a value on channel c. Using this information, the formalism enforces ***minimal waiting*** in our maximal parallelism model by requiring that no pair of processes is ever simultaneously waiting to send and waiting to receive, respectively, on a shared channel.

Definition 3.3 (Wait Records)
$WAIT = \{c! \mid c \in CHAN\} \cup \{c? \mid c \in CHAN\}$.

Let $[\tau_0, \tau_1]$ denote a closed interval of time points; $[\tau_0, \tau_1] = \{\tau \in TIME \mid \tau_0 \leq \tau \leq \tau_1\}$. Then a model, representing a real-time computation of a program, is defined as follows:

Definition 3.4 (Model) Let $\tau_0 \in TIME$, $\tau_1 \in TIME \cup \{\infty\}$, and $\tau_1 \geq \tau_0$. A *model* σ is a mapping $\sigma : [\tau_0, \tau_1] \rightarrow STATE \times \wp(COMM \cup WAIT)$.

Definition 3.5 (Begin/End) For a model σ with domain $[\tau_0, \tau_1]$, define $begin(\sigma) = \tau_0$ and $end(\sigma) = \tau_1$.

Consider a model σ and a point τ with $begin(\sigma) \leq \tau \leq end(\sigma)$. Then $\sigma(\tau) = (state, comm)$ with $state \in STATE$, and $comm \subseteq COMM \cup WAIT$. Henceforth we refer to the two fields of $\sigma(\tau)$ by $\sigma(\tau).state$ and $\sigma(\tau).comm$, respectively. Informally, if σ models a computation of a program S, $begin(\sigma)$ and $end(\sigma)$ denote, resp., the starting time and the termination time of this computation of S ($end(\sigma) = \infty$ if S does not terminate). Furthermore, $\sigma(begin(\sigma)).state$ specifies the initial state of the computation, and if $end(\sigma) < \infty$ then $\sigma(end(\sigma)).state$ gives the final state. In general, $\sigma(\tau).state$ represents the values of program variables at time τ. For a channel name c, the set $\sigma(\tau).comm$ might contain a communication record (c, ϑ), or a wait record of the form $c!$ or $c?$ with the following meaning:

- $(c, \vartheta) \in \sigma(\tau).comm$ if the value ϑ is transmitted along channel c at time τ;
- $c! \in \sigma(\tau).comm$ if S is waiting to send along channel c at time τ;
- $c? \in \sigma(\tau).comm$ if S is waiting to receive along channel c at time τ.

The field $\sigma(end(\sigma)).comm$ will have arbitrary values in the definition of the semantics: the last pair $(state, comm)$ is only used to denote the final state.

In [13] we define a semantic function $\mathcal{M}$ which assigns to each statement S of the programming language a set of models $\mathcal{M}(S)$ such that each model σ in $\mathcal{M}(S)$ represents a possible computation of S starting at any arbitrary time. Here intuition about $\mathcal{M}$ should suffice to motivate the proof system in Section 5.

4 Specifications

To specify functional and timing properties of programs, we use an assertion language which is a real-time extension of temporal logic. In accordance with the semantic model from the previous section, one can refer in our logic at each point of time to the following primitives:

- x to denote the value of program variable x.
- $comm(c, exp)$ to express a communication with value exp along channel c. We also use $comm(c)$ to abstract from the value communicated.
- $wait(c!)$ and $wait(c?)$ to denote that a process is, respectively, waiting to send and waiting to receive along channel c.

As already mentioned in the introduction, our assertion language is real-time version of temporal logic. Traditional linear time temporal logic [10] has been shown to be valuable in the specification and verification of non-real-time systems. It allows the expression of safety and liveness properties by means of a ***qualitative*** notion of time. For instance, for assertions φ, φ_1 and φ_2, this logic contains

- $\Box\varphi$: henceforth φ will hold.
- $\Diamond\varphi$: eventually φ will hold.
- $\varphi_1 \mathbf{U} \varphi_2$ (strong until): eventually φ_2 will hold and until that point φ_1 holds continuously.
- $\varphi_1 \mathcal{U} \varphi_2$ (weak until or unless): either eventually φ_2 will hold and until that point φ_1 holds continuously or φ_1 holds henceforth (in which case φ_2 need never hold).

Furthermore, we have the usual logical connectives such as $\varphi_1 \vee \varphi_2$, $\neg\varphi$, $\varphi_1 \wedge \varphi_2$, and $\varphi_1 \rightarrow \varphi_2$. To give compositional proof rules for sequential composition and iteration, we add the "chop" operator $\mathcal{C}$ and the "iterated chop" operator $\mathcal{C}^*$. Similar operators have been defined in [1] to give a compositional proof system for temporal logic specifications without real-time.

To specify real-time constraints a ***quantitative*** notion of time has to be introduced. In our approach the logic is extended with a special variable T which explicitly refers to the value of a global clock. Intuitively, T refers to the current point of time during execution. Furthermore, we use the special variables

- *start* to express the starting time of a computation and
- *term* to express the termination time of a computation ($term = \infty$ for a non-terminating computation).

In addition to program variables, specifications may also use so-called logical variables which are not affected by program execution. These logical variables can be used to "freeze" the value of a variable at a certain point. E.g., using logical variable v, we can express that eventually x will be incremented by 1: $x := x + 1$ **sat** $x = v \rightarrow \Diamond(x = v + 1)$.

The interpretation of the logic is defined using the computational model of Section 3. To interpret logical variables we use a logical variable environment γ, that is a mapping which assigns a value to each logical variable. First we define the value of an expression exp from the logic in a model σ at time $\tau \geq begin(\sigma)$ and in an environment γ, denoted by $\mathcal{V}(exp)\gamma(\sigma,\tau)$. A few cases from this definition:

- $\mathcal{V}(v)\gamma(\sigma,\tau) = \gamma(v)$
- $\mathcal{V}(x)\gamma(\sigma,\tau) =$
  $\begin{cases} \sigma(\tau).state(x) & \text{if } \tau \leq end(\sigma) \\ \sigma(end(\sigma)).state(x) & \text{if } \tau > end(\sigma) \end{cases}$
- $\mathcal{V}(T)\gamma(\sigma,\tau) = \tau$
- $\mathcal{V}(start)\gamma(\sigma,\tau) = begin(\sigma)$
- $\mathcal{V}(term)\gamma(\sigma,\tau) = end(\sigma)$

The interpretation of an assertion φ at time $\tau \geq begin(\sigma)$ in a model σ and an environment γ is denoted by $\langle\sigma,\tau\rangle\gamma \models \varphi$ and defined by induction on the structure of φ. We give the main clauses.

- $\langle\sigma,\tau\rangle\gamma \models exp_1 = exp_2$ iff
  $\mathcal{V}(exp_1)\gamma(\sigma,\tau) = \mathcal{V}(exp_2)\gamma(\sigma,\tau)$
- $\langle\sigma,\tau\rangle\gamma \models comm(c, exp)$ iff $\tau < end(\sigma)$ and
  $(c, \mathcal{V}(exp)\gamma(\sigma,\tau)) \in \sigma(\tau).comm$
- $\langle\sigma,\tau\rangle\gamma \models comm(c)$ iff there exists a value
  $\vartheta \in VAL$ such that $\langle\sigma,\tau\rangle\gamma \models comm(c,\vartheta)$.
- $\langle\sigma,\tau\rangle\gamma \models wait(c!)$ iff
  $\tau < end(\sigma)$ and $c! \in \sigma(\tau).comm$
- $\langle\sigma,\tau\rangle\gamma \models wait(c?)$ iff
  $\tau < end(\sigma)$ and $c? \in \sigma(\tau).comm$
- $\langle\sigma,\tau\rangle\gamma \models \varphi_1 \mathbf{U} \varphi_2$ iff there exists a $\tau_2 \geq \tau$
  such that $\langle\sigma,\tau_2\rangle\gamma \models \varphi_2$, and
  for all $\tau_1, \tau \leq \tau_1 < \tau_2$: $\langle\sigma,\tau_1\rangle\gamma \models \varphi_1$
- $\langle\sigma,\tau\rangle\gamma \models \varphi_1 \mathcal{C} \varphi_2$ iff σ can be split into models σ_1 and σ_2 with $end(\sigma_1) = begin(\sigma_2) \geq \tau$, $\langle\sigma_1,\tau\rangle\gamma \models \varphi_1$, $\langle\sigma_2, begin(\sigma_2)\rangle\gamma \models \varphi_2$, and $\sigma_1(end(\sigma_1)).state = \sigma_2(begin(\sigma_2)).state$.
- $\langle\sigma,\tau\rangle\gamma \models \varphi_1 \mathcal{C}^* \varphi_2$ iff either σ can be split up into a finite sequence of models such that the last one satisfies φ_2 and all others satisfy φ_1, or σ can be split up into an infinite sequence of models each satisfying φ_1.

In the proof system we will frequently use the following abbreviation. For a finite set $cset$ of channels and directional channels, define
$empty(cset) \equiv \bigwedge_{c \in cset} \neg comm(c) \wedge$
$\bigwedge_{c? \in cset} \neg wait(c?) \wedge \bigwedge_{c! \in cset} \neg wait(c!)$.
As usual, we have $\Diamond\varphi \equiv true\,\mathbf{U}\,\varphi$, $\Box\varphi \equiv \neg\Diamond\neg\varphi$, and $\varphi_1 \mathcal{U} \varphi_2 \equiv (\varphi_1 \mathbf{U} \varphi_2) \vee \Box\varphi_1$.

Let $dch(\varphi)$ denote the set of all c, $c!$, or $c?$ occurring in φ, and $var(\varphi)$ denote the set of all program variables in φ.

Definition 4.1 (Valid Assertion) An assertion φ is *valid*, denoted by $\models \varphi$, iff $\langle\sigma, begin(\sigma)\rangle\gamma \models \varphi$ for any model σ and any environment γ.

Definition 4.2 (Satisfaction) Program S *satisfies* assertion φ, denoted by $\models S$ **sat** φ, iff $\langle\sigma, begin(\sigma)\rangle\gamma \models \varphi$ for any $\sigma \in \mathcal{M}(S)$ and γ.

Finally we give a few simple examples to illustrate our assertion language. General safety properties can be specified, e.g.,

- Program S does not terminate: S **sat** $term = \infty$.
- S does not perform any communication along channel c: S **sat** $\Box\neg comm(c)$.

A few examples of real-time safety properties:

- If S starts its execution with $x = 3$ then S terminates within 6 time units in a state where x has the value 4:
  S **sat** $x = 3 \rightarrow \Diamond(T = term < start + 6 \wedge x = 4)$.
- If S communicates on c then S is waiting to receive or communicating on channel d within 25 time units:
  S **sat** $\Box(T = t \wedge comm(c) \rightarrow$
  $\Diamond(T \leq t + 25 \wedge (wait(d!) \vee comm(d))))$.
  Note that logical variable t is implicitly universally quantified.
- During the execution of S, the program variable x has value 5 at 3 time units after the start of the execution, after 5 time units x has value 8 and y has value 9, and finally after 7 time units program S terminates with $x = 10$ and $y = 12$:
  S **sat** $\Box($ $(T = start + 3 \rightarrow x = 5) \wedge$
  $(T = start + 5 \rightarrow x = 8 \wedge y = 9) \wedge$
  $(T = start + 7 \rightarrow x = 10 \wedge y = 12)) \wedge$
  $term = start + 7$.

Liveness properties can also be expressed:

- S terminates: S **sat** $term < \infty$.
  (Or, equivalently, S **sat** $\Diamond T = term$.)
- S either communicates along channel c infinitely often or eventually it waits forever to send on c:
  S **sat** $(\Box\Diamond comm(c)) \vee (\Diamond\Box wait(c!))$.

5 Proof System

In this section, we give a compositional proof system for our correctness formulas. General well-formedness properties of the semantic model are axiomatized by the following axiom, for any channel c,

Axiom 5.1 (Well-Formedness)

$$S \textbf{ sat } \Box(MW_c \wedge Excl_c \wedge Uniq_c)$$

with

$MW_c \equiv \neg(wait(c!) \wedge wait(c?))$

(*Minimal waiting*: It is not possible to be simultaneously waiting to send and waiting to receive on c.)

$Excl_c \equiv \neg(comm(c) \wedge wait(c!)) \wedge \neg(comm(c) \wedge wait(c?))$

(*Exclusion*: It is not possible to be simultaneously transmitting and waiting to transmit on channel c.)

$Uniq_c \equiv comm(c, exp_1) \wedge comm(c, exp_2) \rightarrow exp_1 = exp_2$

(*Uniqueness*: At most one value is transmitted on channel c at any point of time.)

The next general axiom expresses that a program does not (try to) communicate on channels that do not syntactically occur in the program.

Axiom 5.2 (Communication Invariance)

$$S \textbf{ sat } \Box\, empty(cset)$$

provided $cset \cap dch(S) = \emptyset$.

Similarly, the proof system has an axiom to express that certain program variables are not changed by a program. Let $wvar(S)$ be the set of all write-variables of S, that is, the set of variables occurring in the left-hand side of an assignment or as a variable in an input statement. Obviously, $wvar(S) \subseteq var(S)$.

Axiom 5.3 (Variable Invariance)

$$S \textbf{ sat } x = v \rightarrow \Box x = v$$

provided $x \notin wvar(S)$.

Rule 5.4 (Conjunction)

$$\frac{S \textbf{ sat } \varphi_1,\ S \textbf{ sat } \varphi_2}{S \textbf{ sat } \varphi_1 \wedge \varphi_2}$$

Rule 5.5 (Consequence)

$$\frac{S \textbf{ sat } \varphi_1,\ \varphi_1 \rightarrow \varphi_2}{S \textbf{ sat } \varphi_2}$$

Statement **skip** terminates immediately.

Axiom 5.6 (Skip) **skip sat** $term = start$

The assignment axiom expresses that $x := e$ terminates after K_a time units and that the final value of x equals the value of e in the initial state. If x occurs in the expression e, the initial value of x is needed to evaluate the value of e. This value is recorded by a logical variable v. We use $e[v/x]$ to denote the substitution of each occurrence of x in e by v.

Axiom 5.7 (Assignment)

$x := e$ **sat** $x = v \rightarrow$
$(x = v \textbf{ U } (T = term = start + K_a \wedge x = e[v/x]))$

Example 5.1 We show that we can derive

$x := y + 4$ **sat**
$y = 5 \rightarrow \Diamond(x = 9 \wedge T = term = start + K_a)$.

By the Assignment Axiom and the Consequence Rule we obtain

$x := y + 4$ **sat**
$x = v \rightarrow \Diamond(x = y + 4 \wedge T = term = start + K_a)$.

Since v does not occur in the consequence of this implication, we can substitute x for v.

By the Consequence Rule, this leads to

$x := y+4$ **sat** $\Diamond(x = y+4 \wedge T = term = start+K_a)$.

Since $y \notin wvar(x := y + 4)$, we can derive from the Variable Invariance Axiom

$x := y + 4$ **sat** $y = v \rightarrow \Box y = v$.

Hence, by Conjunction and Consequence,

$x := y + 4$ **sat**
$y = v \rightarrow \Diamond(x = v + 4 \wedge T = term = start + K_a)$.

Since $y = v \rightarrow \Diamond(x = v+4 \wedge T = term = start + K_a)$ implies $y = 5 \rightarrow \Diamond(x = 9 \wedge T = term = start + K_a)$, the Consequence Rule leads to

$x := y + 4$ **sat**
$y = 5 \rightarrow \Diamond(x = 9 \wedge T = term = start + K_a)$.

Statement **delay** e terminates after e time units.

Axiom 5.8 (Delay) **delay** e **sat** $term = start + e$

An output statement starts waiting to send a message, and as soon as a communication partner is available the communication takes place during K_c time units.

Axiom 5.9 (Output)

$c!e$ **sat** $wait(c!)\ \mathcal{U}$
$(T = term - K_c \wedge (comm(c, e) \textbf{ U } T = term))$

An input statement $c?x$ waits to receive a value on the channel c. When the communication takes place the value received is assigned to variable x.

Axiom 5.10 (Input)

$c?x$ **sat** $x = v \rightarrow [(x = v \wedge wait(c?))\ \mathcal{U}$
$(T = term - K_c \wedge (comm(c, x) \textbf{ U } T = term) \wedge (x = v_1 \rightarrow \Box(x = v_1)))]$

Using the $\mathcal{C}$ operator we can easily formulate an inference rule for sequential composition.

Rule 5.11 (Sequential Composition Rule)

$$\frac{S_1 \textbf{ sat } \varphi_1,\ S_2 \textbf{ sat } \varphi_2}{S_1; S_2 \textbf{ sat } \varphi_1\ \mathcal{C}\ \varphi_2}$$

Example 5.2 Consider $x := x+1; x := x+2$. By the Assignment Axiom and Consequence we can derive:
$x := x+1$ **sat** $term = start + K_a \wedge$
$\quad x = v_1 \rightarrow \Box(T = start + K_a \rightarrow x = v_1 + 1)$
and
$x := x+2$ **sat** $term = start + K_a \wedge$
$\quad x = v_2 \rightarrow \Box(T = start + K_a \rightarrow x = v_2 + 2)$.
Then the Sequential Composition Rule and the Consequence Rule lead to
$x := x+1\ ;\ x := x+2$ **sat** $term = start + 2 \times K_a \wedge$
$\quad x = v \rightarrow \Box(T = start + K_a \rightarrow x = v + 1) \wedge$
$\qquad \Box(T = start + 2 \times K_a \rightarrow x = v + 3)$.

Now consider a guarded command G. Define b_G by
$b_G \equiv \bigvee_{i=1}^n b_i$ if $G \equiv [[]_{i=1}^n b_i \rightarrow S_i]$ and
$b_G \equiv \bigvee_{i=1}^n b_i \vee b$ if
$\qquad G \equiv [[]_{i=1}^n b_i; c_i?x_i \rightarrow S_i\ [] \ b; \textbf{delay } e \rightarrow S]$.
Define $Quiet \equiv \bigwedge_{y \in var(G)} y = v \wedge empty(dch(G))$.
First we give an axiom which expresses that there is no activity on the channels of G and no variable of G is changed during the evaluation of the guards. Furthermore we express that if none of the booleans evaluates to true then the guarded command terminates after K_g time units.

Axiom 5.12 (Evaluation)
G **sat** $\bigwedge_{y \in var(G)} y = v \rightarrow$
$(Quiet\ \mathbf{U}\ (T = start + K_g\ \wedge \bigwedge_{y \in var(G)} y = v)) \wedge$
$(\neg b_G \rightarrow term = start + K_g)$

Consider $G \equiv [[]_{i=1}^n b_i \rightarrow S_i]$. If at least one of the booleans yields true then after the evaluation of the booleans one of the statements S_i for which b_i evaluates to true is executed. In the rule we use
$Choice \equiv (term = start + K_g)\ \mathcal{C}\ (\bigvee_{i=1}^n (b_i \wedge \varphi_i))$.

Rule 5.13 (Purely Boolean Guards)

$$\frac{S_i \textbf{ sat } \varphi_i,\ \text{for } i = 1, \ldots, n}{[[]_{i=1}^n b_i \rightarrow S_i] \textbf{ sat } b_G \rightarrow Choice}$$

Next we formulate a rule for
$G \equiv [[]_{i=1}^n b_i; c_i?x_i \rightarrow S_i\ [] \ b; \textbf{delay } e \rightarrow S]$, using
$Wait \equiv \bigwedge_{y \in var(G)} y = v \wedge$
$\quad empty(dch(G) - \{c_1?, \ldots, c_n?\}) \wedge$
$\quad (b \rightarrow T < start + e) \wedge \bigwedge_{i=1}^n (b_i \leftrightarrow wait(c_i?))$,
$InTime \equiv \bigwedge_{y \in var(G)} y = v\ \wedge\ T = term \wedge$
$\qquad (b \rightarrow T < start + e)$,
$EndTime \equiv \bigwedge_{y \in var(G)} y = v\ \wedge\ b \wedge$
$\qquad T = term = start + e$,
$Eval \equiv term = start + K_g$,
$Comm \equiv (Wait\ \mathcal{U}\ InTime) \mathcal{C} \bigvee_{i=1}^n b_i \wedge \varphi_i \wedge comm(c_i)$,
$TimeOut \equiv (Wait\ \mathbf{U}\ EndTime)\ \mathcal{C}\ \varphi$.

Rule 5.14 (IO-guards)

$$\frac{c_i?x_i; S_i \textbf{ sat } \varphi_i,\ \text{for } i = 1, \ldots, n, \quad S \textbf{ sat } \varphi}{[[]_{i=1}^n b_i; c_i?x_i \rightarrow S_i\ [] \ b; \textbf{delay } e \rightarrow S] \textbf{ sat}}$$
$$\bigwedge_{y \in var(G)} y = v \wedge b_G \rightarrow$$
$$Eval\ \mathcal{C}\ (Comm\ \vee\ TimeOut)$$

For an iterated guarded command we have

Rule 5.15 (Iteration)

$$\frac{G \textbf{ sat } \varphi}{\star G \textbf{ sat } (b_G \wedge \varphi)\ \mathcal{C}^*\ (\neg b_G \wedge \varphi)}$$

For parallel composition, we first consider the following simple rule.

Rule 5.16 (Simple Parallel Composition)

$$\frac{\begin{array}{l} S_1 \textbf{ sat } \varphi_1,\ S_2 \textbf{ sat } \varphi_2, \\ \text{neither } \varphi_1 \text{ nor } \varphi_2 \text{ contains } term \end{array}}{S_1 \| S_2 \textbf{ sat } \varphi_1 \wedge \varphi_2}$$

provided $dch(\varphi_i) \subseteq dch(S_i)$ and $var(\varphi_i) \subseteq var(S_i)$, for $i = 1, 2$.

If $term$ occurs in the assertions then we have to take into account that the termination times of S_1 and S_2 are, in general, different. Observe that if S_1 terminates after (or at the same time as) S_2 then the model representing this computation satisfies $\varphi_1 \wedge (\varphi_2\ \mathcal{C}\ (true\ \mathcal{U}\ done))$. Furthermore we have to express that the variables of S_2 are not changed and there is no activity on the channels of S_2 after the termination of S_2. Similarly, for S_1 and S_2 interchanged. Therefore, for $vset \subseteq VAR$, define
$inv(vset) \equiv \bigwedge_{x \in vset} x = v \rightarrow \Box(x = v)$,
$\psi_1 \equiv inv(var(S_2)) \wedge (empty(dch(S_2))\ \mathcal{U}\ T = term)$
$\psi_2 \equiv inv(var(S_1)) \wedge (empty(dch(S_1))\ \mathcal{U}\ T = term)$.
This leads to a general rule for parallel composition:

Rule 5.17 (Parallel Composition)

$$\frac{S_1 \textbf{ sat } \varphi_1,\ S_2 \textbf{ sat } \varphi_2}{S_1 \| S_2 \textbf{ sat } (\varphi_1 \wedge (\varphi_2\ \mathcal{C}\ \psi_1)) \vee (\varphi_2 \wedge (\varphi_1\ \mathcal{C}\ \psi_2))}$$

provided $dch(\varphi_i) \subseteq dch(S_i)$ and $var(\varphi_i) \subseteq var(S_i)$, for $i = 1, 2$.

In [13] we have proved the soundness of our proof system, that is, every formula S **sat** φ which can be proved in our proof system is valid. Furthermore we have established relative completeness, that is, if valid formulas from our assertion language can be proved, then any valid formula S **sat** φ can also be derived in our axiomatic system.

6 Conclusion

We have formulated a sound and relatively complete proof system to verify that a program satisfies a specification written in a version of explicit clock temporal logic. Important is that our axiomatization is compositional, and hence allows us to reduce the complexity of the verification problem. In contrast with for instance [11], we have not required a discrete time domain. Often a dense time domain is convenient, since it allows events to be arbitrary close to each other in time. This is important when reactive systems are modeled. Moreover, by having dense time we can easily reason about the timing of atomic actions on one level of abstraction and their refinement into several actions on a lower, more concrete, level. A close look at the development of the theory reveals that this choice is orthogonal to the rest of the theory. Therefore, should for other reasons, e.g., efficient automatic verification, a discrete domain appear more attractive, then this can be substituted without causing changes to the theory elsewhere.

As mentioned in the introduction, the proof system from this paper is closely related to the compositional axiomatization given in [7]. We feel that after the present exposition we may expand a little more on the relation to that paper. Apart from the use of a simpler programming language without program variables, the main difference is that a quantitative treatment of time is obtained by the addition of explicit bounds to the modal operators. Observe that $\varphi_1 \mathbf{U}_{<\tau} \varphi_2$ in metric temporal logic is equivalent to $T = t \wedge (\varphi_1 \mathbf{U}(T < t + \tau \wedge \varphi_2))$ in explicit clock temporal logic. From this translation it becomes clear that in MTL all timing properties are specified relative to a starting point, whereas explicit clock temporal logic directly refers to the value of the clock. In future work we will address the precise relation between these two approaches and investigate, by means of examples, which version is most applicable for specific applications.

References

[1] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proc. ACM Symp. on Theory of Computing*, pages 51–63, 1984.

[2] E. Emerson, A. Mok, A. Sistla, and J. Srinivasan. Quantitative temporal reasoning. presented at *Workshop On Automatic Verification Methods For Finite State Systems*, Grenoble, 1989.

[3] E. Harel. Temporal analysis of real-time systems. Master's thesis, The Weizmann Institute of Science, Rehovot, Israel, 1988.

[4] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit clock temporal logic. In *Proc. Symp. Logic in Computer Science*, pages 402–413, 1990.

[5] J. Hooman. A denotational real-time semantics for shared processors. In *Parallel Architectures and Languages Europe*, volume II, pages 184–201. LNCS 506, Springer-Verlag, 1991.

[6] J. Hooman. *Specification and Compositional Verification of Real-Time Systems.* PhD thesis, Eindhoven University of Technology, 1991.

[7] J. Hooman and J. Widom. A temporal-logic based compositional proof system for real-time message passing. In *Parallel Architectures and Languages Europe*, volume II, pages 424–441. LNCS 366, Springer-Verlag, 1989.

[8] INMOS Limited. OCCAM *2 Reference Manual*, 1988.

[9] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[10] Z. Manna and A. Pnueli. Verification of concurrent programs: a temporal proof system. In *Foundations of Computer Science IV, Distributed Systems: Part 2*, volume 159 of *Mathematical Centre Tracts*, pages 163–255, 1982.

[11] J. Ostroff. *Temporal Logic for Real-Time Systems.* Advanced Software Development Series. Research Studies Press, 1989.

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

[13] P. Zhou, J. Hooman, and R. Kuiper. A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness. Technical report, Eindhoven University of Technology, 1991.