Interface models

Ravn, Anders P.; Staunstrup, Jørgen

Published in:
Proceedings of the Third International Workshop on Hardware/Software Codesign

Link to article, DOI:
10.1109/HSC.1994.336711

Publication date:
1994

Document Version
Publisher's PDF, also known as Version of record

Interface Models

Anders P. Ravn and Jørgen Staunstrup

Department of Computer Science,
Technical University of Denmark,
DK–2800 Lyngby, Denmark
e-mail: {apr,jst}@id.dtu.dk

Abstract

This paper proposes a model for specifying interfaces between concurrently executing modules of a computing system. The model does not prescribe a particular type of communication protocol and is aimed at describing interfaces between both software and hardware modules or a combination of the two. The model describes both functional and timing properties of an interface.

1 Motivation

The design of non-trivial computing systems requires a module concept to structure the design into manageable subparts. Modules can be newly created for a specific purpose, but often they are general purpose off-the-shelf components. In both cases it is important that the interface to the module is simple and well documented. This paper describes a model for defining module interfaces covering both off-the-shelf components and new components.

In a design of a sequential program, a module is characterized by a signature that introduces names for constants, datatypes, operations, etc. A database module may for example have an interface like

```
MODULE database;
  TYPE
    Key    = 1..MaxKey
    Elem   = RECORD k: Key; ... END
    Answer = ...

    DB     = ...
    ...
  OPERATION
    insert: DB x Key x Elem -> DB x Answer
    find:   DB x Key        -> Answer
END
```

Such a concept is present in modular or object oriented programming languages, in specification languages like VDM [4] or Z [15], and also in hardware description languages like VHDL [8] etc. In some of the languages, the signature is extended with constraints that specify how the operations are related and what results can be expected. There are many syntactical variations of this module concept, but a common goal is to structure a system into manageable parts.

However, there are very few computing systems that can be described as one run of a single sequential program. Most systems are reactive, the operations are performed in some order determined by communications with an environment. Thus, each module may specify constraints on the order of operations. This paper thus focuses on fundamental concepts that allows module specifications with a succinct description of the communication interface.

The notions of module and computing systems are interpreted in a very general sense encompassing combinations of hardware and software and models on different levels of abstraction from primitive building blocks to complex sub-systems.

A common source of errors and delays in design and development projects is misunderstanding caused by ambiguous interface descriptions. At the same time it is unrealistic (at least in the foreseeable future) to insist that designers give complete formal specifications of all aspects of a module. Our attitude is to leave it to the designer what to specify rigorously, but to provide a model where a wide variety of different aspects of an interface can be studied unambiguously in particular the timing aspects.

Even a superficial analysis of existing interfaces reveals a large variation. Therefore, the proposed model does not prescribe a particular communication discipline, instead it allows a variety of disciplines to be described rigorously. Examples of existing disciplines are the synchronous message passing dictated by (soft-

0-8186-6315-4/94 $04.00

ware) models such as CSP [7], CCS [11], LOTOS [1], etc., and the signaling disciplines of (hardware) models in VHDL. Such specialized disciplines have significant advantages later in a design process where they permit various forms of analysis and optimization. However, we think that the variation found in module interfaces makes it unrealistic to prescribe a particular rigid discipline during development and specification of a system.

2 The model

This section proposes an interface model following the guidelines discussed in the introduction. A module encapsulates a part of a computation. For the purposes of this paper, it is not necessary to go into the details of the computational model of a module, it is assumed to be a state machine where the state is represented by a number of state variables. A state variable may denote a simple boolean or a composite value like the database mentioned earlier. A state is changed by applying given operations, e.g., `find` or `insert` on the state variables.

We would like to stress that an interface specification is *not* intended as a complete specification of all aspects of a module. Specification of types and operations may be left open.

The interface of a module consists of one or more state variables that are shared with other modules. In addition there can be local (hidden) state variables in each module. Sharing implies that the value of the state variables in the interface may be changed both by local computations and computations by other modules. A state based model is used here in contrast to the message passing approach found in CSP [7] and CCS [11], but in agreement with UNITY [2] and TLA [10]. We have found a state based model useful for a wide variety of purposes [16] on different levels of abstraction ranging from circuits to computer networks.

A state variable x is modeled by a function from *Time* to its range of values, e.g., if x is boolean, we write

$$x : \mathit{Time} \rightarrow \mathbf{Bool}$$

and if x is the database mentioned in the introduction

$$x : \mathit{Time} \rightarrow \mathtt{DB}$$

The *Time* domain is taken to be the non-negative reals.

Example: a simple database server

Consider an application where the database is placed in a server. A simple protocol defines how a client can request that an operation is performed on the database and how the server answers the request.

For simplicity, we consider only one client and request-answer messages without data. This interface is modeled by two state variables

$$\mathit{req} : \mathit{Time} \rightarrow \mathbf{Bool}$$
$$\mathit{ans} : \mathit{Time} \rightarrow \mathbf{Bool}$$

An extension to multiple clients could be modeled by indexing the variables, and data is modeled by substituting a composite type for the simple type **Bool**.

The server interface uses a simple (four-phase) protocol where the values of the interface variables change in the following sequence:

$$\ldots \mathit{req} \ldots \mathit{ans} \ldots \neg\mathit{req} \ldots \neg\mathit{ans} \ldots \mathit{req} \ldots$$

It is important that the client module referencing the interface obey the same protocol. It is therefore important to enable designers to document/specify protocols in a rigorous manner. However, just giving the sequencing above leaves many open questions, e.g.,

- Does an *ans* always follow a *req*?
- How long time is the *ans* enabled?

To answer such questions one must specify the protocol by a predicate on the interface. The interpretation of this predicate is that modules referencing an interface may assume that the associated predicate holds, on the other hand any module changing the values of state variables in the interface are obliged to ensure that the resulting values satisfy the predicate. At this point the notation for specifying such interface predicates is not defined. It will at least include propositional logic over relations formed by operations on state variables (for the variables above boolean expressions using operators such as $\wedge, \vee, \neg$, etc). This would allow the specification of time invariant properties.

The idea to model states as functions of continuous time is well known in most fields of engineering and science. However, it is associated with the theory of dynamic systems which uses differential or difference equations to express properties of systems. Such equations give deterministic specifications (functions) while we want to preserve some choices for the designers. Therefore, in section 3, we introduce a notation

allowing the designer to state a large variety of properties including timing constraints and other dynamic properties in a constraint oriented manner.

Before going into specification of dynamic properties, another important aspect of an interface protocol is discussed.

2.1 Definedness

In any non-trivial interface, the values of state variables change; however, these changes may not always be atomic transitions from one well-defined state to another. It is often the case that a state change goes through some intermediate values that should be ignored. At a low level, digital signals do not change instantaneously from one binary value to another. A similar phenomenon can be observed at higher levels where composite values (integers, vectors, records, a database, etc.) may change in a piecewise fashion, and hence go through some values where the interpretation is not defined. In order to model this, we assume that there is a predicate def_x associated with every state variable, x, of an interface. The intended interpretation is that def_x holds whenever x has a well-defined value, whereas no assumptions can be made about x when def_x does not hold.

In many cases, definedness involves timing. For example, in a typical digital circuit, it is only meaningful to inspect the values of state variables during some phases of the global clock signal. However, the *def* predicate can be illustrated without involving timing aspects.

Example: a simple server (continued)

Consider a lower level model of the server where the interface is modeled with signals which may change continuously within some range: $[l..h]$. This could, for example, model a voltage level.

$$req', ans' : Time \rightarrow [l..h]$$

Furthermore, assume that signal values less than a certain threshold value V_F are interpreted as the boolean value *false* and signal values larger than another threshold value V_T, ($V_T > V_F$) are interpreted as the boolean value *true*, i.e.

$$true(v) \equiv v > V_T \quad \text{and} \quad false(v) \equiv v < V_F$$

The *def* predicate for the interface variable ans' is

$$def_{ans'} \equiv true(ans') \vee false(ans')$$

Any client using the server is obliged to ensure that $def_{ans'}$ holds whenever the value of ans' is used. No assumptions can be made about the behavior of ans' when $def_{ans'}$ is false.

end of example

To summarize, interfaces are documented with protocols and definedness predicates. The protocols specify constraints that must be met both by the module and by the environment. Definedness describes when it is safe to interpret the state variables of the interface. When the definedness predicate does not hold, the state variables are allowed to be in inconsistent with a protocol. For example, when a physical signal represented by a voltage level is changing it might temporarily have values that cannot consistently be interpreted as high or low. Definedness is used to filter out such intermediate values of interface variables.

3 Timing

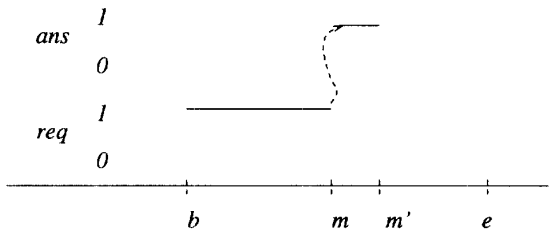
There are many proposals for specifying timing properties of protocols. Some advocate the use of explicit states that model time or local clocks, others incorporate it into a temporal logic, and yet others extend process algebras with timing constructs. For an introduction, we recommend the proceedings [5, 17]. However, if we look at practice, protocols are almost exclusively defined by timing diagrams. For that reason we prefer to use an interval logic, Duration Calculus [3] that is a symbolic notation for reasoning about timing properties. It has been used to reason successfully about phases of computations in the design of real-time systems [14], in hybrid systems [18], in circuits [6] and it has been embedded into the Z specification language [9].

Example: a simple server (cont.)

Part of the protocol specification for the server might be:

If the request signal is maintained for 10 time units then answer will come.

This is illustrated by the following timing diagram:



We will write this part of a protocol as

$$P_1: \quad (\lceil req \rceil \wedge \ell = 10) \longrightarrow \lceil ans \rceil$$

This formula is interpreted over the state and an interval $[b, e]$ of *Time*. The symbol $\longrightarrow$ is a "followed-by" operator which means that if the interval can be split into an initial subinterval $[b, m]$ where the precondition $\lceil req \rceil \wedge \ell = 10$ holds then the interval $[m, e]$ will have an initial subinterval $[m, m']$ where the postcondition $\lceil ans \rceil$ holds.

The formula $\lceil req \rceil$ holds on an interval exactly when the state *req* holds almost everywhere in the interval (this formulation avoids stating anything about the value of *req* in individual points of a dense time domain, e.g., at the end points of the interval). The formula $\ell = 10$ holds when the interval has a length of 10, and conjunction $\wedge$ has its conventional meaning. Informally, the formula, P_1, reads: "If *req* holds for an interval of length 10 then it is followed by *ans* holding". Note that nothing is said about *ans* if *req* holds in a shorter interval, this means that *ans* may or may not hold for such an interval.

3.1 Reasoning about protocols

A formula, such as the protocol stated above, is no more than a symbolic representation of a timing diagram. A particular formula holds for a given collection of state variables, just when it holds for any subinterval of *Time*. This corresponds with the intuition that a timing diagram is a snapshot of the state trajectory.

One reason for using a formalism, rather than the intuitive diagrams, is that the formulas allow us to calculate consequences of a given protocol. The Duration Calculus has a number of general rules for doing such calculations. The appendix gives a brief overview of the Duration Calculus. As an example, we will calculate the assumptions needed to ensure that *ans* holds for at least 5 time units. For this we need an inference rule[13]

$$\frac{(\lceil x \rceil \wedge \ell = t) \longrightarrow \lceil y \rceil}{(\lceil x \rceil \wedge \ell = t + t') \Rightarrow (\ell = t \,;\; (\lceil y \rceil \wedge \ell = t'))}$$

where t, t' are positive reals.

The inference rule has a formula above the line stating an assumption, that is sufficient to ensure the conclusion, which is the formula occurring below the line. In the conclusion the subformula, $\lceil x \rceil \wedge \ell = t + t'$ characterizes an interval of length $t + t'$ where x holds almost everywhere. The other subformula,

$$\ell = t \,;\; (\lceil y \rceil \wedge \ell = t')$$

uses the chop-operator: "; ". This formula holds of an interval $[b, e]$ just when it can be split into an interval $[b, m]$ where $\ell = t$ holds and an interval $[m, e]$ where $\lceil y \rceil \wedge \ell = t + t'$ holds.

The general inference rule can be instantiated to a particular case, for example, by replacing x with *req*, y with *ans*, t' with 5, and t with 10. This means that when the protocol P_1 is assumed then

$$(\lceil req \rceil \wedge \ell = 15) \Rightarrow (\ell = 10 \,;\; (\lceil ans \rceil \wedge \ell = 5))$$

In other words, if *req* holds for 15 time units, then *ans* is sure to hold for the last 5 time units.

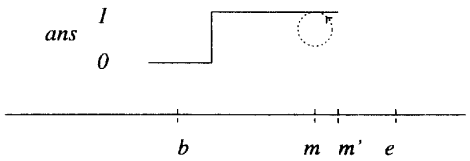
Note that the protocol specification does not prescribe any particular behavior if *req* is held for less than 10 time units. The server is then free to ignore it or act upon it.

Example: a simple server (cont.)

A protocol may also include stability constraints, for instance to ensure that a definedness predicate can be checked or a value can be moved to a local state. An example is:

If the ans signal is set, then it is stable for at least 10 time units.

illustrated by the following timing diagram:



or as a formula

$$S_1: \quad ((\lceil \neg ans \rceil \,;\, \lceil ans \rceil) \wedge \ell \leq 10) \longrightarrow \lceil ans \rceil$$

The subformula $\lceil \neg ans \rceil \,;\, \lceil ans \rceil$ characterizes an interval where *ans* is set, i.e., changes from false to true.

The formula thus reads: "If *ans* is set within an interval of length less than 10 then it continues to be set".

Another example is that the server for some reason stabilizes the period before an answer. This is expressed by a stability formula stating that *ans* is first set after 5 time units of *req*

$$S_2: \ ((\lceil \neg req \rceil \ ; \ \lceil req \rceil) \wedge \ell \leq 5) \longrightarrow \lceil \neg ans \rceil$$

end of example

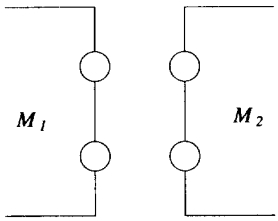
A protocol is in general specified by a conjunction of progress formulas like P_1 and stability formulas like S_1 and S_2. The protocol constrains the state trajectories of the system. A state trajectory that for any point of time satisfies the protocol, i.e., where the protocol formulas hold for any subinterval, is acceptable.

4 Composition and feasibility

Specification of a protocol by conjoining formulas requires some care. The result should be *feasible*, that is, the combined protocol should have at least one trajectory which is consistent with all formulas. This is not always the case when protocols are pieced together. Infeasibility often occurs when one module with a feasible protocol is composed with another with a different feasible protocol. The result might very well be an infeasible protocol.

Example: composition with a server (cont.)

Consider the composition of the the server M_1 with a client M_2.



Assume that the client for some reason insists on releasing the answer after 2 time units.

$$P_2: \ (\lceil \neg req \rceil \wedge \ell = 2) \longrightarrow \lceil \neg ans \rceil$$

An initial state trajectory for the server could be characterized by

$$\lceil \neg req \rceil \ ; \ (\lceil req \rceil \wedge \ell = 10) \ ; \ (\lceil \neg req \rceil \wedge \ell = 5)$$

We can then use the server request protocol P_1 and its two stability protocols S_1 and S_2 to calculate that for this trajectory it is also the case that

$$\ell > 10 \ ; \ (\lceil \neg req \wedge ans \rceil \wedge \ell = 5)$$

The client release protocol P_2 would allow us to calculate that for the same trajectory it is the case that

$$\ell > 12 \ ; \ (\lceil \neg ans \rceil \wedge \ell = 3)$$

The conjunction of the two formulas specify a trajectory with a final subinterval of length 3 where both *ans* and $\neg ans$ holds. This is obviously impossible.

end of example

During the design and development of a modular system, the modules are considered separately and while developing a particular module, the designer formulates protocols characterizing the interface. Later, the separately developed modules are combined and at this point the consistency of the associated protocols must be ensured. Formally, this is done by checking the feasibility of composing the protocols.

4.1 Feasibility

This section states general results on feasibility which can be used to check that the composition of protocols remain feasible.

Feasibility can be ensured by inspecting the initial and goal conditions of "followed-by" formulas that are conjoined. This gives rise to the following three sufficient conditions for feasibility. They are stated as theorems (the proofs are given in [13] but omitted here for brevity).

The first theorem ensures that mutually exclusive progress properties can be combined:

Theorem 1 *A conjunction*

$$\bigwedge_{i=0}^{m} (\lceil P_i \rceil \wedge \ell = t_i) \longrightarrow \lceil P_i' \rceil$$

defines a feasible system when each goal P_i' is pointwise satisfiable, and the collection of initial conditions $(P_i, i = 0, \ldots, m)$ are mutually disjoint.

The condition that the goal must be satisfiable is necessary to exclude strange protocols like

$$\lceil req \rceil \longrightarrow \lceil \neg req \wedge req \rceil$$

This theorem ensures that the composition of the release protocol P_2 and the request protocol P_1 is feasible, because the predicates *req* and $\neg req$ are disjoint.

If the initial conditions are non exclusive, feasible progress properties must either cooperate on the goals or there must be a winner in a race. This is expressed in the second theorem.

Theorem 2 *A conjunction of progress commitments*

$$((\lceil P_1 \rceil \wedge \ell = t_1) \longrightarrow \lceil P_1' \rceil) \wedge$$
$$((\lceil P_2 \rceil \wedge \ell = t_2) \longrightarrow \lceil P_2' \rceil)$$

with $t_1 \leq t_2$, defines a feasible system when each goal P_i' is satisfiable and when either $P_1' \wedge P_2'$ is satisfiable or $P_1' \Rightarrow \neg P_2$.

The condition $P_1' \Rightarrow \neg P_2$ ensures that the P_1 wins over P_2 when they are simultaneously enabled.

The second theorem is important if *req* has a side effect on some new state variable, say

$$(\lceil req \rceil \wedge \ell = 4) \longrightarrow \lceil busy \rceil$$

or if we go for automatic release after 5 time units

$$(\lceil req \wedge ans \rceil \wedge \ell = 5) \longrightarrow \lceil \neg req \rceil$$

There is a similar result linking stability and progress formulas. Essentially, they agree (composing them is feasible) when either the initial conditions are exclusive, the goals cooperate, or the progress time is not smaller than the stability time.

These theorems are stated to sketch the possibilities offered by the Duration Calculus. There is work in progress on automating this calculus (at least with suitable assumptions about the time domain) and this will hopefully enable us to provide tools which can be used for checking the consistency of interfaces.

5 Conclusion

This paper discusses the importance of interface specifications which are rigorous but incomplete specifications of a module. Two important aspects of such interface specifications are protocols and definedness predicates. Furthermore, timing is an important part of an interface, and it has been shown how to formalize the well known timing diagrams and use this formalization for specifying an interface.

Acknowledgment

We would like to thank Jens Ulrik Skakkebæk, Zhiming Liu, and Michael R. Hansen for their comments to an earlier version of this paper.

This work is partially supported by the Commission of the European Communities (CEC) under the ESPRIT programme in the field of Basic Research Action proj. no. 7071: "ProCoS II", and by the Danish Technical Research Council under the "Codesign" programme.

References

[1] E. Brinksma. *On the Design of Extended LOTOS.* Technical University Twente, Holland, 1988. Dissertation.

[2] K. Mani Chandy and Jajadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[3] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Proc. Letters*, 40(5), December 1991.

[4] J. Dawes. *The VDM-SL reference guide.* Pitmann, 1991.

[5] J. W. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors. *Real-Time: Theory in Practice, REX Workshop*, volume 600 of *LNCS.* 1992.

[6] M.R. Hansen, Z. Chaochen, and J. Staunstrup. A real-time duration semantics for circuits. In *Proceedings TAU 1992 ACM/SIGDA Workshop on Timing Issues in Specification and Synthesis of Digital Systems*, 1992. Princeton, NJ, March 18-20, 1992.

[7] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[8] IEEE, New York. *VHDL Language Reference Manual*, std 1076-1987 edition, 1988.

[9] R. Inal. Modular specification of real-time systems. In *Proc. 6th Euromicro Workshop on Real-Time Systems.* IEEE Computer Society Press, 1994.

[10] L. Lamport. The temporal logic of actions. Technical report, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, USA, 25 December 1991.

[11] R. Milner. *Communication and Concurrency.* Series in Computing Science. Prentice-Hall, 1989.

[12] B. Moszkowski. A temporal logic for multi-level reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.

[13] A. P. Ravn. Design of embedded real-time computing systems. Manuscript, May 1994.

[14] A.P. Ravn, H. Rischel, and K. M. Hansen. Specifying and verifying requirements of real-time systems. *IEEE Trans. Software Engineering*, 19(1):41–55, Jan. 1993.

[15] J. M. Spivey. *The Z Notation.* Prentice-Hall, 1989.

[16] Jørgen Staunstrup. *A Formal Approach to Hardware Design.* Kluwer Academic Publishers, 1994.

[17] J. Vytopil, editor. *Proceedings Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *LNCS*. 1991. Nijmegen 6-10 Jan. 1992.

[18] Chaochen Zhou, A. P. Ravn, and M. R. Hansen. An extended duration calculus for hybrid real-time systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 36–59, 1993.

Appendix: Overview of Duration Calculus

A system is described by a collection of named *state variables* which are functions of *Time*, modeled by the real numbers. Properties of systems are expressed by constraining the state variables over time. We wish to express requirements and design without explicit mentioning of particular time instants, and introduce a notation which is a real-time, interval logic based on state durations.

Syntax

We assume the names of state variables $X, \ldots$ together with their *value domains* $Type_X, \ldots$ given by declarations in a suitable specification language. The language should comprise names for constants and operators. The type **R** of real numbers should be available with the usual operators and so should the type **Bool** of boolean values with the usual propositional operators. We use lower case names $a, b, x, \ldots$ to denote *rigid variables* of any type in the language. Rigid variables denote time-independent entities.

State expressions and state assertions

The set of *state expressions* is generated by

1. every constant, rigid variable, or state variable is a state expression,
2. any type of correct expression $op(S_1, \ldots, S_n)$ formed from an operator symbol op and state expressions $S_1, \ldots, S_n$ is a state expression.

A *state assertion* is a state expression with type **Bool**.

Durations and duration terms

For any state assertion P, $\int P$ is a *duration*. A *duration term* is of type **R** and is generated by

1. durations, real constants and rigid variables are duration terms,
2. if op is an n-ary operator symbol over **R** and $r_1, \ldots, r_n$ are duration terms, then $op(r_1, \ldots, r_n)$ is a duration term.

The symbol ℓ is used as an abbreviation for the duration term $\int true$.

Duration formulas

If A is any n-ary predicate symbol on **R** and $r_1, \ldots, r_n$ are duration terms, then $A(r_1, \ldots, r_n)$ is an *atomic duration formula.* A *duration formula* is of type **Bool** and generated by

1. atomic duration formulas and the symbols *true* and *false* are duration formulas,
2. if $\mathcal{D}_1$ and $\mathcal{D}_2$ are duration formulas, so are $(\neg\mathcal{D}_1)$, $(\mathcal{D}_1 \vee \mathcal{D}_2)$, $(\mathcal{D}_1 \ ; \ \mathcal{D}_2)$, and $(\forall x)\mathcal{D}_1$ where x is a rigid variable.[1]

We use standard abbreviation $\wedge, \Rightarrow, \Leftrightarrow$ for both state assertions and duration formulas, and we introduce abbreviations for commonly used duration formulas

Abbrev.	**Formula**	**Legend**
$\lceil\ \rceil$	$\ell = 0$	point
$\lceil P \rceil$	$\int P = \ell \wedge \ell > 0$	almost everywhere P
$\Diamond \mathcal{D}$	$true \ ; \ \mathcal{D} \ ; \ true$	somewhere $\mathcal{D}$
$\Box \mathcal{D}$	$\neg(\Diamond\neg\mathcal{D})$	always $\mathcal{D}$
$\mathcal{D} \longrightarrow \lceil P \rceil$	$\neg\Diamond(\mathcal{D} \ ; \ \lceil\neg P\rceil)$	P follows D

for state assertion P and duration formula $\mathcal{D}$.

[1] In [3] the "chop"-operator ";" [12] is denoted by "$\frown$".1

For duration formulas, the following rules of precedence are used

first:	$\neg$, $\Box$, $\Diamond$
second:	$\vee$, $\wedge$, ;
third:	$\Rightarrow$, $\longrightarrow$

The logical operators are overloaded: They are used for state assertions as well as duration formulas; but the meaning can be inferred from the type of the operands, e.g., "$\vee$" denote disjunction for state assertions in "$\lceil P_1 \vee P_2 \rceil$" and disjunction for duration formulas in "$\lceil P_1 \rceil \vee \lceil P_2 \rceil$".

Semantics

A (particular) behavior $\mathcal{B}$ of a system assigns a meaning to each state variable X as a function

$$X : [0, \infty) \rightarrow Type_X$$

giving the state as function of time from the start time $t = 0$.

For a given behavior, we select a value $\mathcal{V}(x)$ for each rigid variable x. Each state expression then denotes a function on $[0, \infty]$ (where the value is obtained by evaluating the expression for each point of time, and where each rigid variable denotes a constant function with value $\mathcal{V}(x)$).

An observation *interval* is a closed and bounded interval $[b, e] \subset [0, \infty)$. For a given interval, the duration $\int P$ of a state assertion P denotes the real number

$$\int_b^e (\textbf{ if } P(t) = tt \textbf{ then } 1 \textbf{ else } 0)\, dt$$

which is the measure of the set of points in $[b, e]$ where P is true. For any interval $[b, e]$, duration terms and atomic duration formulas denote real and boolean values.

The values of composite duration formulas $\neg\mathcal{D}$, $\mathcal{D}_1 \vee \mathcal{D}_2$ and $(\forall x)\mathcal{D}$ on $[b, e]$ are obtained by the usual interpretation of the logical operators and quantification. The value of a "*chop*" formula $\mathcal{D}_1 \,;\, \mathcal{D}_2$ is true if and only if the interval $[b, e]$ can be divided into two parts $[b, m]$ and $[m, e]$ (with $b \leq m \leq e$) such that $\mathcal{D}_1$ is true on $[b, m]$ and $\mathcal{D}_2$ is true on $[m, e]$).

A duration formula $\mathcal{D}$ *holds* on the interval $[b, e]$ for the behavior $\mathcal{B}$ just when $\mathcal{D}$ is true on $[b, e]$ with any assignment $\mathcal{V}$ of values to the rigid variables.

The formula $\mathcal{D}$ *holds from start* on the behavior $\mathcal{B}$ just when it holds on any interval of the form $[0, T]$ for the behavior $\mathcal{B}$.

A duration formula $\mathcal{D}$ is *valid* (a tautology) just when it holds for every interval $[b, e]$. It is sufficient for a formula to be valid that it holds from start for every behavior $\mathcal{B}$.

Finite variability

The interpretation of a state variable X may be confined to certain classes of functions, e.g., continuous or differentiable functions. For a state variable X with a discrete value domain $Type_X$ it is reasonable to demand *finite variability*: Any interval $[b, e]$ can be divided into finitely many subintervals with X constant on each (open) subinterval.