

Project Number: 033471

SPEEDS

Speculative and Exploratory Design
in Systems Engineering

Integrated Project

Information Society Technologies

D.2.5.4 Contract Specification Language (CSL)

Due date of deliverable: April 1st 2008

Actual submission date: April 1st 2008

Start date of project: May 1st 2006

Duration: 36 months

Israel Aerospace Industries

Revision: SPEEDS-RE-D.2.5.4.a-20080401

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	V
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

PROPRIETARY RIGHTS STATEMENT

THIS DOCUMENT CONTAINS INFORMATION, WHICH IS PROPRIETARY TO THE SPEEDS CONSORTIUM. NEITHER THIS DOCUMENT NOR THE INFORMATION CONTAINED HEREIN SHALL BE USED, DUPLICATED OR COMMUNICATED BY ANY MEANS TO ANY THIRD PARTY, IN WHOLE OR IN PARTS, EXCEPT WITH THE PRIOR WRITTEN CONSENT OF THE SPEEDS CONSORTIUM THIS RESTRICTION LEGEND SHALL NOT BE ALTERED OR OBLITERATED ON OR FROM THIS DOCUMENT. THE RESULTS PRESENTED IN THIS TECHNICAL REPORT WILL BE PUBLISHED ELSEWHERE. HENCE THIS TECHNICAL REPORT SHALL NOT BE USED AS A REFERENCE FOR THESE RESULTS.

Abstract

This is an internal deliverable. It describes Patterns-CSL, a version of CSL (Contracts Specification Language) where contracts' assertions are specified by pre-defined patterns of common assertions that are instantiated by user's parameters. Instantiated patterns will be translated into ESM, SPEEDS core formalism. The document includes description of the various kinds of parameters that can be used in patterns, and preliminary list of patterns.

Overall writing: Vered Gafni (IAI). Essential contributions from: Albert Benveniste (INRIA), Benoit Caillaud (INRIA), Susanne Graf (VERIMAG), and Bernhard Josko (OFFIS).

Document History

Version	Date	Modification
v.0.1	5/3/2008	Beginning document
v.0.2		Adding "defined-by" construct. Correcting typo errors. Adding explanations required by users.
v.0.3	8/6/2009	Extend the Proprietary Statement by Bernhard Josko

LIST OF CONTENTS

1 INTRODUCTION	4
2 HRC SPECIFICATION	4
2.1 Variables Specification and Behaviors	5
2.1.1 Conditions	6
2.1.1.1 Derived Conditions and Special pwc variables	6
2.1.2 Events.....	7
2.1.2.1 Derived Events.....	8
2.1.2.2 Event Expression	8
2.1.3 Intervals	9
2.2 Contracts Specification	10
2.3 Assertions' Specification.....	11
2.3.1 Patterns.....	12
2.3.2 Pattern instantiation	13
2.3.2.1 Free values	13
2.3.2.2 Related Intervals	14
2.3.3 Patterns Specification	15
3 CSL GRAMMAR	21

LIST OF FIGURES

<i>FIGURE 1: HRC SPECIFICATION STRUCTURE</i>	4
<i>FIGURE 2: BEHAVIOR EXAMPLE</i>	5
<i>FIGURE 3: A BEHAVIOR OF A PWC-CONTINUOUS VARIABLE</i>	5
<i>FIGURE 4: A BEHAVIOR OF A DISCRETE PWC VARIABLE</i>	6
<i>FIGURE 5: A BEHAVIOR OF A PURE EVENT</i>	6
<i>FIGURE 6: A TIMER BEHAVIOR</i>	7
<i>FIGURE 7: THE BEHAVIOR OF A PERIODIC TIMER VARIABLE</i>	7
<i>FIGURE 8: AN EVENT AND ITS DISCRETE-PWC TRANSFORMATION</i>	7
<i>FIGURE 9: ILLUSTRATION OF OCCURRENCES OF THE EXPRESSIONS: $E_1 \wedge E_2$, $E_1 - E_2$, AND $E_2 - E_1$</i>	9
<i>FIGURE 10: OCCURRENCES OF THE SEQUENCE EVENT</i>	9
<i>FIGURE 11: EXAMPLE OF OCCURRENCES OF THE INTERVAL $[A, B]$ LABELED BY THE PRECEDENCE ORDER</i>	10
<i>FIGURE 12: EXAMPLE OF A SLIDING WINDOW OCCURRENCES</i>	10
<i>FIGURE 13: CONSTRAINED BEHAVIORS OF ROOM TEMPRATURE</i>	11
<i>FIGURE 14: CONTRACT RELATIONS</i>	11
<i>FIGURE 15: SATISFACTION RELATION: $M \models C \Leftrightarrow_{DEF} A \cap M \subseteq P$</i>	11
<i>FIGURE 16: EXAMPLE OF ITERATIVE OCCURRENCES' INSTANCES</i>	12

1 Introduction

CSL – Contracts Specification Language - is intended as a pragmatic proposal for specifying an HRC by contracts. CSL provides for HRC specification (*Figure 1*), namely: its interface (the I/O and internal state-variables' signatures), and the associated contracts (but not the implementation). A contract consists of a pair of assertions classified as assumption and promise {A/P}, respectively. In general assertions specify constraints over the behaviors of the interface variables as explained in the subsequent subsections. Formally, assertions are expressed by ESM, the underlying formalism of SPEEDS.

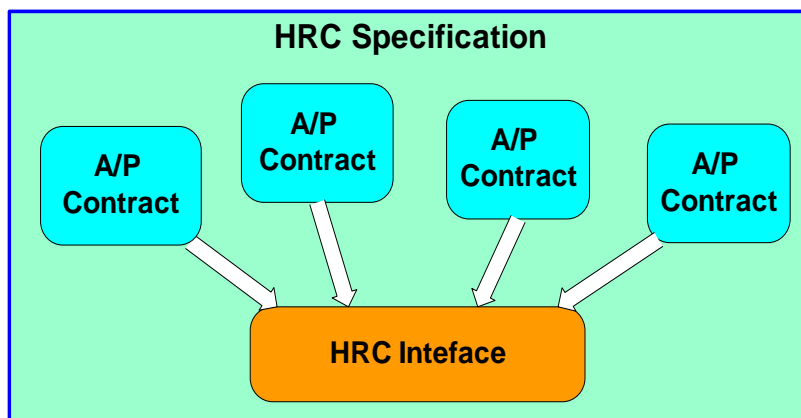


Figure 1: HRC Specification Structure

Whilst all the rest is a matter of decision of some syntax, the core problem of industrial users is really a friendly means for formal specification of assertions, namely ESMs. During project meetings, three forms for expressing assertions in CSL were discussed: Patterns, Visual automata, and Extended PSL.

This document describes the "patterns" version of CSL. A pattern is a textual expression embedded with placeholders for parameters; it represents a function (in software terms) that returns an ESM corresponding to a certain assertion. The assertion represented by a concrete pattern is fixed up to parameters' instantiation.

2 HRC Specification

The general scheme of an HRC specification is presented below¹. Here **bolded** terms denote reserved names/symbols, 'HRC-Id' is the HRC identifier, *italic* terms denote objects to be expanded, and * denotes a list of definitions of this type (possibly 0).

HRC {HRC-Id}

Interface

Input: {*variable definition**}

Output: {*variable definition**}

Internal: {*variable definition**}

Contracts

{*contract specification**}

¹ BNF presentation of CSL grammar is provided in Section 3.

2.1 Variables Specification and Behaviors

A variable declaration is represented in the form: *variable-id: variable-type*, e.g. $x:Integer$. The *variable-type* specifies its domain set of values; in general all common types are supported by CSL. As means for modeling physical systems, variables get values along periods or discrete instants of time during the system operation. We assume continuous physical time represented by $\mathbb{R}_{\geq 0}$ - the nonnegative real axis. Let T_x denote the time domain of a variable $x:D_x$. (namely: $T_x \subseteq \mathbb{R}_{\geq 0}$ comprises the time intervals/instants where x is defined). Then, every function $f_x: T_x \rightarrow D_x$ represents a possible behavior of x (Figure 2), and the set of all such functions, $T_x \mapsto D_x$, define the space of behaviors of x , denoted by B_x

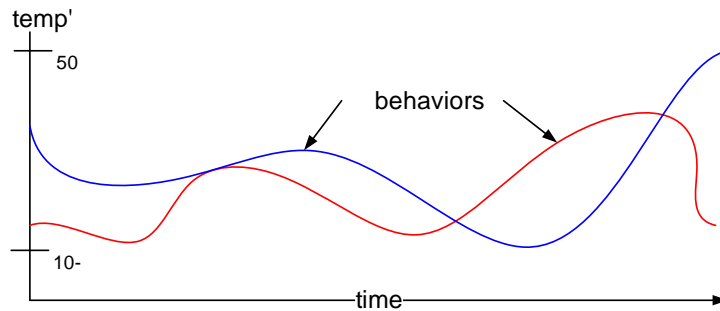


Figure 2: Let the variable $RoomTemp: [-10.0..50.0]$ represent room temperature that can range from -10° to 50° . So the space of behaviors for this variable is $\mathbb{R}_{\geq 0} \mapsto [-10.0..50.0]$ - the set of all functions from $\mathbb{R}_{\geq 0}$ to the specified domain. The figure above illustrates 2 possible behaviors of this set.

CSL allows 2 kinds of variables that are classified according to their time domains: *Piecewise-Continuous (pwc)* and *events*².

- *Piecewise-Continuous (pwc)* are variables that are defined and evolve continuously along some time interval (bounded, or the entire time axis), or enumerable concatenation of adjacent time intervals (assuming proper open/closed correspondence of the meeting points) with possibly step-discontinuity at the meeting points. Two kinds of pwc-variables are further classified by their domains as:
 - *Continuous-pwc* variables are those whose values' domain is continuous (Figure 3). A concrete behavior of such a variable is specified by the time and value at the beginning of each interval, and a function that describes its behavior along the interval.

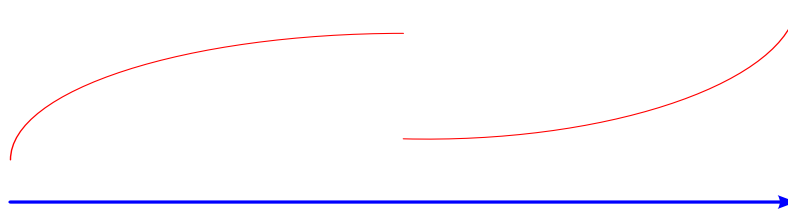


Figure 3: A behavior of a pwc-continuous variable

- *Discrete-pwc* variables are those whose values domain is discrete, namely their values over each interval is constant (Figure 4). Thus, a concrete behavior of such a variable is specified by the time and value at the beginning of each interval, solely.



² Pwc generalizes the continuous and discrete kinds defined in the MM, and event corresponds to the same kind of interaction-point/pin defined by the MM.

Figure 4: A behavior of a discrete pwc variable

- A *pure event* is a single valued variable that is defined only at an ordered (possibly denumerable but necessarily divergent) sequence of discrete time instants (Figure 5). In general, consecutive event occurrences may "stutter" at certain time instants – namely: have the same time stamp but still ordered - but this is limited to finite many occurrences.



Figure 5: A behavior of a pure event

Since it is single valued, the actual value of a pure event is insignificant and the only interesting information is the fact of its occurrence. So, a pure event is said to *occur* (or be present) at a time instant t if it is defined there; otherwise it does not occur (or absent) at t .

An event, in general, may be associated with a finite domain of values declared as a normal variable $e:\{v_1, \dots, v_n\}$, called *valued-event*. A valued-event is actually considered as a declaration of a set of mutual exclusive pure events $\{e=v_1, \dots, e=v_n\}$ where $e=v_i$ denotes the occurrences of the e when it gets the value v_i .

2.1.1 Conditions

Condition is a Boolean (discrete) pwc variable or a Boolean expression constructed of Boolean variables using the normal connectors: \vee , \wedge , \neg , \rightarrow , \leftrightarrow . The domain of a Boolean expression is the intersection of the domains of the constituting variables, and the interpretation is by normal semantics of Boolean connectors, namely: for Boolean variables a , b ,

- $a \vee b$ is True at every instant where a is True or b is True, and False otherwise.
- $a \wedge b$ is True at every instant where a is True and b is True, and False otherwise.
- $\neg a$ is True at every instant where a is False.
- $a \rightarrow b$ is True at every instant where a is False, or a is True and b is True, and False otherwise (namely: $a \rightarrow b$ is equivalent to $\neg a \vee b$).
- $a \leftrightarrow b$ is True at every instant where a and b are simultaneously True or False, and False otherwise.

2.1.1.1 Derived Conditions and Special pwc variables

Derived conditions define conditions by predicates over non-Boolean variables. The general form of a derived condition is $x \sim \mathbf{exp}$ where \sim is an order relation ($<$, \leq , \geq , $>$), and \mathbf{exp} is a relation over variables that must be well defined in the context of the underlying variables. For instance, the condition $x > 5$ is defined only if 5 is in the range of x , and is True wherever x indeed satisfies the relation, and false at the complementary sub-intervals (w.r.t. domain of x). Similarly, $x < y$ is defined only if x and y share the same range.

CSL provides for some special derived conditions, as follows.

- An expression of the form: $x \sim \mathbf{F} \mathbf{init} v$ where F is a pwc function over *pwc* variables such that $F(0)=0$, and v is a value in the range of x . Such an expression is considered a condition that is

true at every interval $[a,b)$ where $x(t)=F(t-a)+v$ for $a\leq t<b$.

In this context, CSL provides predefined *timer* functions, namely: continuous-pwc variables with an initial value 0 and constant derivative that equals 1 over every interval in its domain. Timers are used mainly to introduce delays and periodic operation that are specified as follows.

- An expression **Timer(T) at e** defines a timer on every interval that starts with an occurrence of e and ends either after T time-units if e didn't reoccur meanwhile, or otherwise at the occurrence of e , in which case it restarts evolving along a new interval; the timer is 0 everywhere else (*Figure 6*).

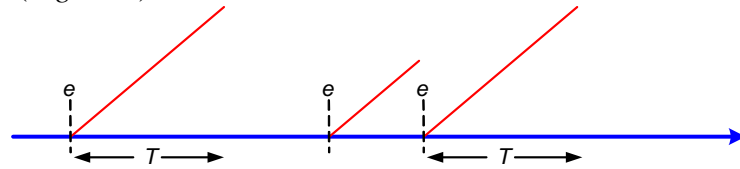


Figure 6: A timer behavior

- An expression **PeriodicTimer(T) at e** defines a timer that starts operating at the first occurrence of e and then periodically restarts evolving from 0 whenever reaching the value T (necessarily every T time-units - *Figure 7*).

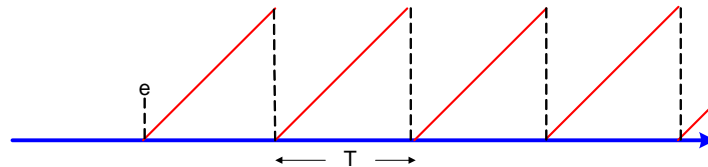


Figure 7: The behavior of a periodic Timer variable

CSL assumes a basic physical time-unit whose value (μsec , second, minute, etc.) is defined by the user per application. Time specifications, T , are expressed in terms of Natural number of the basic time-unit. (e.g., 10 seconds).

- An expression **pwc(e) init v** is a kind-transformer from valued events to discrete-pwc variables that is derived from e by sustaining its values between its occurrences (the instants where it is defined), and gets the value v until first occurrence (*Figure 8*).

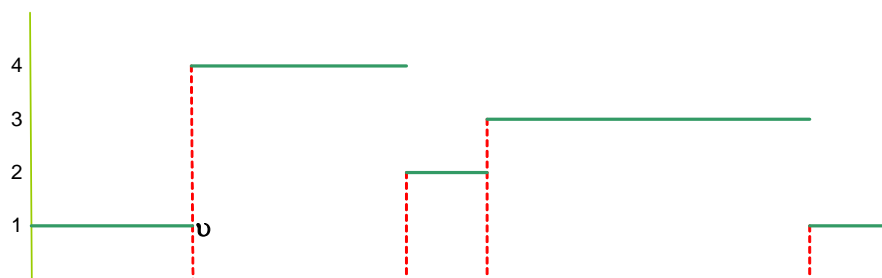


Figure 8: An event and its discrete-pwc transformation

- For any discrete-pwc variable x , the derived variables **prev(x)**, **next(x)** get the value of x at the previous/next interval, respectively (with proper initialization of **prev(x)**).

2.1.2 Events

In addition to application primitive events, and predefined system events (e.g., **Startup** that occurs just once as the system starts operating), CSL provides for definition of derived events

and event expressions, as described below.

2.1.2.1 Derived Events

Derived events are events that are defined in terms of occurrences of state changes of the system. A state change occurs with any change of value of any variable. We restrict derived variables, however, to denote only discrete changes, thus avoiding dense occurrences.

A derived event definition takes the form of: ***e* defined-by state-change**. In general, a state-change is specified by a pair of predicates: (*Pre*, *Post*) each specifies a set of states (allowed assignments to variables); the event *e*, thus defined, occurs at every instant where there is a change from a *Pre*-state to a *Post*-state. CSL provides the following pre-defined derived variables (in the sequel, we will support a more general definition by explicit *Pre*, *Post* predicates).

- Derived from a pwc variable *x*, the declaration *event(x)* defines an event that occurs whenever a discrete change in the value of *x* takes place.
- Derived from a condition *c*, *tr(c)*, *fs(c)* denote those instants where *c* becomes true or false, respectively (including start-up). Based on this mechanism, CSL provides special expressions for timeout and periodic events as follows.
 - An event (*e+T*) assumes an implicit definition of a timer *c:=Timer(T)* at *e*, and corresponds to the event defined by *tr(c=T)*; namely it occurs *T* time units after last occurrence of *e* (recall an occurrence of *e* during *T* resets the timer).
 - An event *periodic(e,T)* assumes an implicit definition of a timer *c:=PeriodicTimer(T)* at *e*, and corresponds to the event defined by *tr(c=T)*; namely: it occurs every *T* time units starting from first occurrence of *e*. We shall use the term *periodic(T)* as a shorthand for *periodic(Startup,T)*.

2.1.2.2 Event Expression

Event expressions express relations that result in filtering or joining separate occurrences of given events. Event expressions are defined recursively as follows.

- Every event identifier is an event expression, where for a valued event *e*: $\{v_1, \dots, v_n\}$ the event identifier *e* denotes the event $(e=v_1) \vee \dots \vee (e=v_n)$ (see below for \vee -expressions). Also, the following are event expressions.
 - For a valued event *e*, ***e rel v*** where ***rel*** is any arithmetic relation ($<$, \leq , $>$, \geq) and *v* is in the domain of *e* (e.g., $e > 5$). ***e rel v*** selects the occurrences of *e* where the relation ***rel*** is satisfied.
 - For an event *e* and a condition *C*, the expression ***e when C*** defines an event that occurs at every time instant that *e* occurs provided *C* is defined and evaluates to True at that time instant.
- An expression constructed from events by Boolean operators as follows ($\{e\}$ is the set of occurrence-instants of an event *e*):

$$\{e_1 \wedge e_2\} = \{e_1\} \cap \{e_2\}, \{e_1 \vee e_2\} = \{e_1\} \cup \{e_2\}, \{e_1 - e_2\} = \{e_1\} - \{e_2\}$$

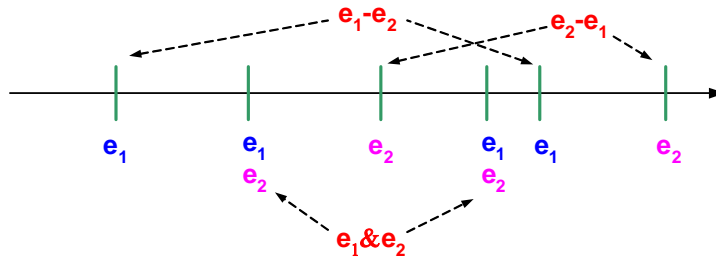


Figure 9: Illustration of occurrences of the expressions: $e_1 \wedge e_2$, $e_1 - e_2$, and $e_2 - e_1$ (note $e_1 \vee e_2$ occurs whenever e_1 occurs or e_2 occurs)

In addition, we define the “sequence” operator, denoted by “;”, such that for any events e_1, e_2 the event $e_1; e_2$ occurs at the first occurrence of e_2 (strictly³) after each occurrence of e_1 (Figure 10); an occurrence of e_2 that follows a number of occurrences of e_1 defines a single occurrence of $e_1; e_2$.

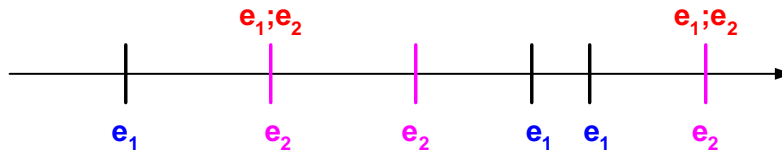


Figure 10: occurrences of the sequence event

The operator “;” is associative thus we write $e_1; e_2; \dots; e_n$ to denote a composite event whose occurrences require a sequence of events to occur. Also, the term $n\#e$ is used as an abbreviation for the expression $e; \dots; e$ ($n > 1$ times). Note that consecutive occurrences of $n\#e$ necessarily rely on partially overlapping occurrences of the sequence $e; \dots; e$.

2.1.3 Intervals

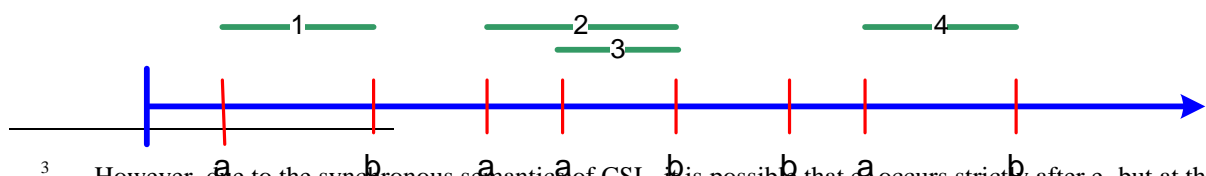
Intervals denote durations confined by occurrences of events. Intervals may take any form of left/right close/open end points denoted by the standard notation for intervals: $[...]$, $(...)$, $(...]$, $[...)$, respectively; in what follows the notation $|...|$ is used to denote any kind of beginning/end points.

The concrete syntax of an interval specification in CSL is $|e_1, e_2, \dots, e_n|$ where e_1, e_2, \dots, e_n are events, and $n \geq 1$. The term $n:e$ is used as an abbreviation for the expression e, \dots, e (n times).

An interval expression $|e_1, e_2, \dots, e_n|$ occurs with every occurrence of the event $e_1; e_2; \dots; e_n$ and lasts from the occurrence of e_1 till the occurrence of e_n . Note, however, that since the occurrences of e_1 and e_n may coincide (although order is preserved) this definition might yield void intervals. In particular, for $n=1$ only closed singleton intervals, $[e_1]$, may be specified.

Occurrences of intervals are totally ordered by the order of the occurrences of the beginning events, namely, a concrete occurrence of an interval $|a_1, a_2, \dots, a_n|$ precedes a concrete occurrence of an interval $|b_1, b_2, \dots, b_n|$ if and only if the occurrence of a_1 precedes the occurrence of b_1 (

Figure 11).



³ However, due to the synchronous semantics of CSL, it is possible that e_2 occurs strictly after e_1 but at the same time instant.

Figure 11: Example of occurrences of the interval $[a,b]$ labeled by the precedence order.

In applications, intervals usually consist of 2-length sequence of events: $[a,b]$, longer sequences, however, are also useful. For instance, an interval of the form $|n:e|$ specifies a "sliding window" of n occurrences of e (Figure 12). In particular, when e is a periodic time event this form defines a fixed length sliding time window. In such a case, the syntax $|n+1:tu|$ is relaxed to $|n tu|$ to specify a sliding window of n time-units. Such a specification implicitly assumes the existence of a periodic event, tu that ticks every time-unit. For instance, the specification $[3sec]$ defines a sliding window of 3 seconds where it assumes a periodic event declaration "sec defined-by *periodic(1sec)*"

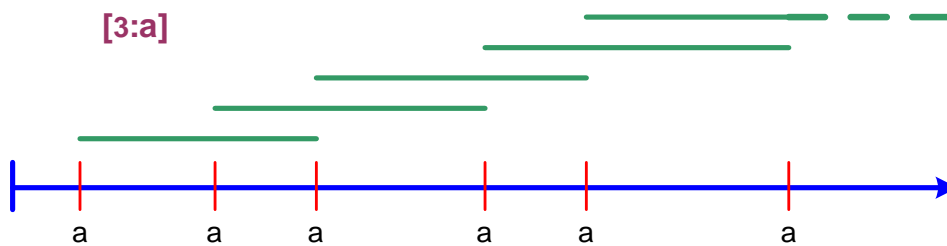


Figure 12: Example of a sliding window occurrences

2.2 Contracts Specification

A contract is identified by name and an attribute designating its view-point. The content of a contract consists of a pair of assertions classified as *assumption* and *promise*, respectively. The general form of a contract specification is:

{viewpoint-id} **contract** {contract-id}

Assumption: {assertion}

Promise: {assertion}

Here, contract-id is a unique name identifying the contract. The viewpoint attribute, designated by viewpoint-id, indicates an aspect of the system. CSL provides a list of pre-defined viewpoints' names, such as: *functionality*, *performance*, *timing*, *safety*, etc. A viewpoint has no formal semantics but is used as a means of sorting contracts across a complete system specification (probably consists of a numbers of HRC modules) for the purpose of analysis.

The specification of the assumption and promise assertions is actually the core of the contract; it presents a required capability of the component (associated with the viewpoint). Let C be a component (HRC) defined with the interface variables x_1, \dots, x_n and let B_x denote the set of all possible behaviors of x (Section 2.1) then the Cartesian product $B_C = B_{x_1} \times \dots \times B_{x_n}$ defines the behaviors' space for that component. An *assertion* over x_1, \dots, x_n constrains some (possibly all) of the variables' behaviors thus constraining, the possible behaviors of the component to a subset of B_C . For instance, the assertion "after 5 seconds, room temperature is always between 25° and 30°" constrains the behaviors of the room temperature (Figure 2) to a subset presented graphically in Figure 13.

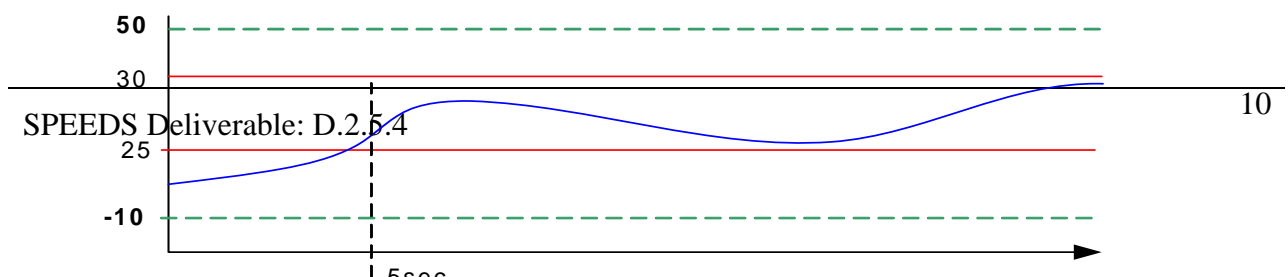


Figure 13: Constrained behaviors of Room temprature.

An *assumption* is an assertion that specifies given behaviors of the component (guaranteed by other components, or the environment), the component does nothing to produce them, and cannot refuse them. Assumptions normally refer to the input variables (constrain their behaviors). A *promise*, on the other hand, is an assertion that specifies behaviors the component is responsible to produce. Promises normally express relations between input and output or local variables; or just constraints on the behaviors of output or local variables.

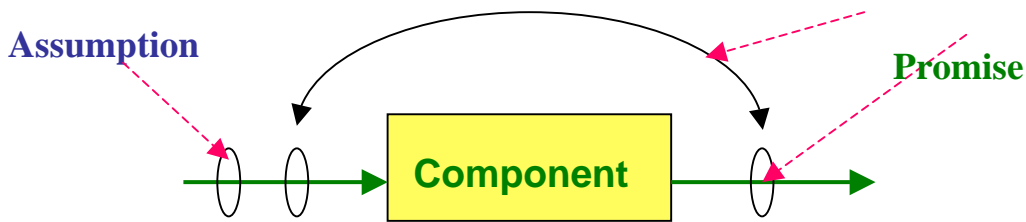


Figure 14: Contract relations

The classification of contracts' assertions into assumption and promise categories does not affect the specification of these assertions but plays a crucial role in the analysis of the HRC specification. For instance, consistency check will look to validate that a contract assumptions are promised by other contracts (within the HRC, other HRC, or the environment). Similarly, satisfaction of a contract by an implementation (Figure 15) verifies that the set of behaviors produced by the implementation that are also consistent with the assumption indeed forms a subset of the set of behaviors promised by the contract.

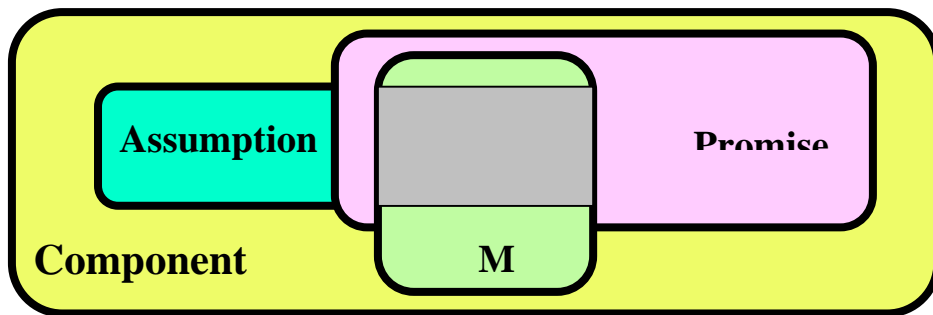


Figure 15: Satisfaction relation: $M \models C \Leftrightarrow_{def} A \wedge M \subseteq P$

2.3 Assertions' Specification

A contract assertion, either assumption or promise, is specified in CSL as a set of atomic assertions each one expressed by a *pattern*; the contract assertion is considered to be the conjunction of the atomic assertions.

A pattern is a textual expression that represents a function (in software terms) that returns an ESM corresponding to a certain assertion. The assertion represented by a concrete pattern is

fixed up to parameters' instantiation. Parameters can take form of events, conditions, and time intervals as defined in Section 2.1. For instance, if one would like to express the assertion: whenever the request button is pressed a car should arrives at the station within 3 minutes, she could use the pattern:

whenever [E: event] **occurs** [E: event] **occurs** *within* [I: interval]

instantiated by concrete propositions that in this case would look like:

whenever [car-request] **occurs** [car-arrives] **occurs** *within* [3min]

Patterns, in general, express reactive behavior of the form: "triggering behavior implies promised behavior" which means: whenever the triggering behavior is identified at some time instant or interval, the promised behavior will occur at some time instant or interval right now or in bounded future. For instance, in the above instantiated pattern "car-request" is the triggering behavior, and "car-arrives within 3min" is the promised behavior. An *occurrence-instance* of the behavior of a reactive pattern refers to the duration that starts with an identification of the triggering behavior and ends with the following occurrence of the promised behavior.

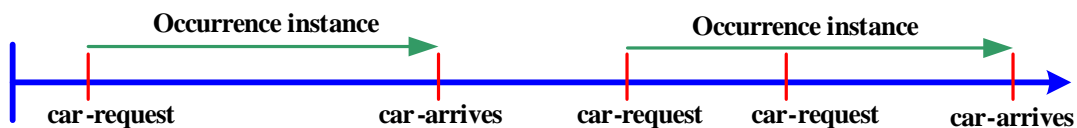


Figure 16: Example of iterative occurrences' instances

A pattern can be used in two manners, depending on the evaluation of its occurrence-instances:

- In the "iterative" manner (Figure 16) only iterative occurrence-instances of the behavior are considered, namely: next identification of triggering behavior does not start before the end of the promised behavior of previous instance (hence, a triggering behavior that occurs while an occurrence-instance takes place is ignored).
- In the "flowing" manner, every occurrence-instance of the behavior is considered.

In practice, most of the patterns' usage is of the iterative manner. Flowing interpretation is significant only when there is one-to-one matching between occurrences of the triggering behavior and the promised behavior; for instance, when the promised behavior is expected to occur fixed duration after the triggering behavior.

2.3.1 Patterns

The syntax of a pattern consists of a fixed text strings (that hint its purpose) embedded with place holders for parameters. A placeholder is denoted by the type of parameter that is allowed to be instantiated, enclosed by square brackets. A pattern's parameter is of one of the following types: *event*, *condition*, *interval*, and a *Natural* number. Actual placeholders' typing takes one of the following forms:

- [E] denotes a placeholder that should be instantiated by *event* expression, solely.
- [C] denotes a placeholder that should be instantiated by *condition* expression, solely.
- [S] denotes a placeholder that can be instantiated an expression that is either an event, or a condition.
- [I] denotes a placeholder that should be instantiated by an interval expression, solely.
- [N] denotes a placeholder that should be instantiated by a Natural number.

2.3.2 Pattern instantiation

Normal pattern instantiation takes place by filling each placeholder in the pattern by a concrete expression according to the proper parameter type. CSL allows, however, some "syntactic sugar" extensions of parameters' specification that express relations between objects (events/conditions) in, possibly separate, placeholders; these provide for compact specifications that otherwise, by normal expressions, would become somehow cumbersome.

2.3.2.1 Free values

A free value is an identifier that is associated with a finite domain (such as domains of discrete-pwc variables, events, or ranges of indices). A free value has global scope over all placeholders' instantiations in a pattern. Namely: a certain free value may appear with several variables, in separate placeholders within a pattern instantiation; in every interpretation of the pattern instantiation, however, all the occurrences of a free value are considered to denote the same value (all variables that refer to that free value must have a common domain over which the free value is defined).

Free values are used in 2 forms: universal and existential quantification.

1. The universal form is implicitly specified just by usage of a free value. Every occurrence of a universal quantified free variable is considered to represent a set of patterns' instantiations, one for each possible value of the free variable.
2. The Existential form, on the other hand, must be explicitly indicated by a term "**exists x.**" where "exists" is a reserved word and x is the identifier of the free value. Every occurrence of an existential quantified free variable is considered to represent an expression in the instantiation that is disjunction of the set of all possible values in all occurrences. For instance, the term **exists v. X=v and Y=v** stands for the expression: $(X=v_1 \text{ and } Y=v_1) \text{ or } \dots \text{ or } (X=v_n \text{ and } Y=v_n)$ where X and Y share the domain $\{v_1, \dots, v_n\}$.

The actual usage and specification of free values is demonstrated in the following examples.

1. Given event definitions: $e:\{v_1, \dots, v_n\}$, let $x=\text{pwc}(e)$ **init 0**, namely: x is discrete-pwc expansion of e . To express this property by an assertion we may use the pattern

whenever [E] occurs [C] holds during following [I]

instantiated as follows.

whenever $[e=v]$ occurs $[x=v]$ holds during following $[e;e]$

This instantiation is interpreted as the set of normal pattern's instantiations.

{ **whenever $[e=v_1]$ occurs $[x=v_1]$ holds during following $[e=v_1;e]$,**
...
whenever $[e=v_n]$ occurs $[x=v_n]$ holds during following $[e=v_n;e]$ }

2. Let **CarLocation[1:400] StationLocation[1:250][1:250]** denote the present location of the U-cars and the locations of the stations, respectively. Then to express the assumption: *At startup every car is at a station*, we may use the pattern **At Startup [C]** with the instantiation:

At Startup [CarLocation[m]= exists k. StationLocation[k]]

This instantiation contains a universal quantified value m that ranges over the number of cars, and an existential quantified value k the ranges over the number of stations, Hence, it defines 400 patterns' instantiations (due to m) each of the form:

At Start-up [CarLocation[#]=StationLocation[I] or ...

or CarLocation[#]=StationLocation[6250]]

3. Let **CarDestination[1:400]** denote the present destination of the U-cars (same domain as stations). Then to express the promise: *Every U-car reaches its destination within 5 minutes*, we may use the pattern **every [E] is followed by [S] within following [I]** with the instantiation:

whenever [CarDestination[m]=(i,j) occurs [CarLocation[m]=(i,j)] occurs within [5min]

This instantiation contains 2 universal quantified values: m that ranges over the number of cars, and the pair (i,j) that range over station locations; summing up to representation of 2500000 patterns' instantiations.

4. Free values can be also used to specify relations among the associated variables. For instance, mutual exclusion can be expressed by the following instantiation in the pattern **Always [C]**:

Always [Busy[j] and Busy[k] $\rightarrow j=k$]

2.3.2.2 *Related Intervals*

In a reactive pattern the promise constraint usually relates to the triggering event. For instance, in the pattern:

whenever [car-request] occurs [car-arrives] occurs within [car-request, car-request+3min]

the 3 minutes constraint refers to the triggering event *car-request*. In case like that CSL allows to omit the triggering event in the interval specification; it is implicitly assumed. Thus, the compact form of the above pattern instantiation would be:

whenever [car-request] occurs [car-arrives] occurs within [3min]

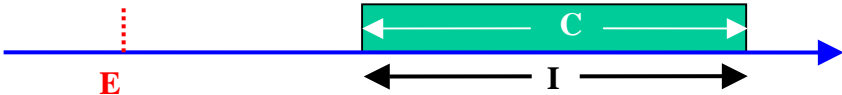
The rule in general is when e is the triggering event of a pattern:

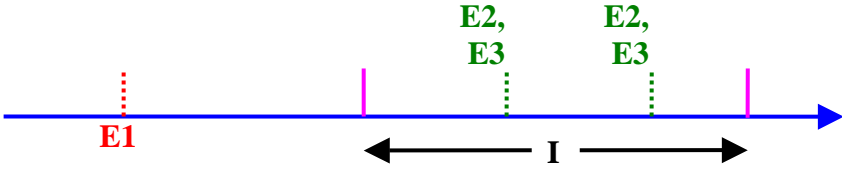
- Interval specification $[t_1, t_2]$ denotes the interval expression $[e+t_1, e+t_2]$
- Interval specification $[t]$ denotes the interval expression $[e, e+t]$

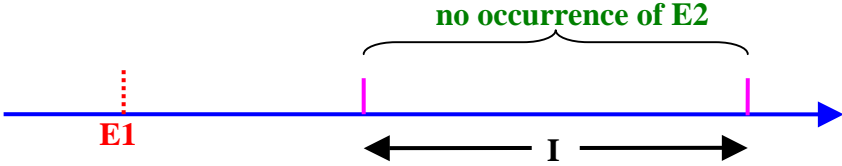
2.3.3 Patterns Specification

In what follows, we present a set of patterns for what is supposed to be most frequent kind of assertions that are used in contracts specification.

The presentation of patterns below includes, for each pattern, informal intended semantics, usage interpretation: iterative or flowing, examples, and also – when applicable - derived patterns that are actually syntactic sugar, namely: they represent special cases of core patterns that deserve particular attention (thus need not special ESM implementation but can be substituted with the proper pattern).

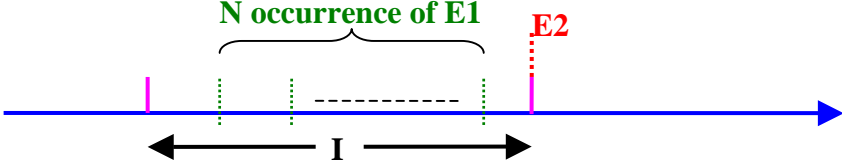
Pattern Id	P1	
Name	whenever [E] occurs [C] holds during following [I]	
Description	<p>Following every occurrence of E, C will hold during the following occurrence of the interval I. The occurrence-instances are defined recursively by the first occurrence of E and the expiration time of following I.</p> 	
Usage	Iterative	
Examples	<p>- The gate is closed when a train traverses gate region (GR). whenever [EnterGR] occurs [Gate=closed] holds during following [EnterGR, ExitGR]</p> <p>- During first 10 seconds of movement, car's velocity dynamics is $x' = -kx + U$ whenever [CarStartsMoving] occurs [$x' = -k*x + U$ init($x=0$)] holds during following [10s]</p>	
<i>Derived patterns</i>		
[E] implies [C] holds forever	whenever [E] occurs [C] holds during following [E,false]	
Always [C]	[Startup] implies [C] holds forever	
Whenever [E] occurs [C] holds	whenever [E] occurs [C] holds during following [0]	
At Startup [C]	whenever [Startup] occurs [C] holds during following [0]	

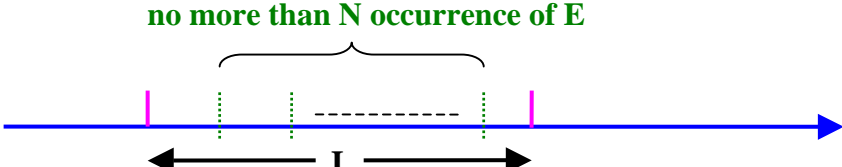
Pattern Id	P2	
Name	whenever [E1] occurs [E2] implies [E3] during following [I]	
Description	<p>An occurrence instance of this pattern is triggered by the occurrence of E1 and ends with the termination of the following occurrence of I. In response, at each time instant during the following occurrence of the interval I at which E2 occurs, E3 occurs as well. Note that it is not possible to use a term like $E2 \rightarrow E3$ for events since the negation of events is not defined in our formalism (this is in contrast to conditions).</p> 	
Usage	iterative	
Examples	<p>- <i>Dispatching commands will be refused during first 3 seconds after a car arrives at station</i> whenever [car-arrives] occurs [dispatch-cmd] implies [refuse-msg] during following [3sec]</p>	
<i>Derived Patterns</i>		
[E1] implies [E2] during following [I]	whenever [True] occurs [E1] implies [E2] during following [I]	

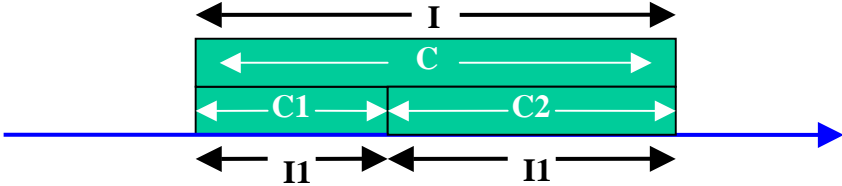
Pattern Id	P3
Name	whenever [E1] occurs [E2] does not occur during following [I]
Description	<p>Following every occurrence of E1, E2 will not occur during the following occurrence of the interval I. The occurrence-instances are triggered by the occurrence of E1 and ends with the termination of the following occurrence of I.. Note that it is not possible to use a term like $\neg E2$ since the negation of events is not defined in our formalism (this is in contrast to conditions).</p> 
Usage	Iterative
<p>- Example: 40 sec. minimal delay between trains: - whenever [Tin] occurs [Tin] does not occur during following (40sec]</p>	
<i>Derived Patterns</i>	
[E1] cannot occur before [E2]	whenever [StartUp] occurs [E1] does not occur during following [StartUp,E2)

Pattern Id	P4
Name	whenever [E] occurs [S] occurs within [I]
Description	<p>Following every occurrence of E, S (that is either an event or a condition) will hold at some time within the following occurrence of the interval defined by I. Occurrence instance is defined by the occurrence of E and the following occurrence of S that falls in I.</p>
Usage	Iterative
Examples	<p>- It takes a train 15 to 25 seconds to traverse gate area. whenever [EnterGR] occurs [ExitGR] occurs within (+15sec,+25sec]</p> <p>- when an occupied car arrives at a station, all passengers leave the car within 5 sec. whenever [CarArrives when CarStatus=occupied] occurs [CarStatus=Empty] occurs within (+5sec]</p>

Pattern Id	P5
Name	[C] during [I] raises [E]
Description	<p>If C holds during I then E occurs at the end point of I</p>
Usage	flowing
Example	<p>- A fail signal is emitted whenever inlet flow is less than L for 3 seconds continuously [Fin' < L] during [3sec] raises [FailSignal]</p>

Pattern Id	P6
Name	[E1] occurs [N] times during [I] raises [E2]
Description	<p>If E1 occurs at least N times during I then E2 occurs at the end point of I</p> 
Usage	flowing
Example:	<p>- A fail signal is emitted whenever an error occurs 3 times or more within 5 sec.</p> <p>[Error] occurs [3] times during [5sec] raises [FailSignal]</p>

Pattern Id	P7
Name	[E] occurs at most [N] times during [I]
Description	<p>The event E occurs at most N times during I</p> 
Usage	flowing
Example:	<p><i>Between the time an elevator is called at a floor and the time it stops at that floor the elevator can pass that floor at most twice.</i></p> <p>[PassFloor[m]] occurs at most [2] times during (CallAtFloor[m], StopAtFloor[m])</p>

Pattern Id	P8
Name	[C] during [I] implies [C1] during [I1] then [C2] during [I2]
Description	<p>Whenever C holds during I, there exists a partition $I = (I1; I2)$ such that C1 holds during I1 and C2 holds during I2.</p> 
Usage	iterative
Examples	<p>Examples</p> <ul style="list-style-type: none"> - On a straight line, a U-car shall accelerate at a rate of $2m/sec^2$ until reaching velocity of $30m/sec$, maintain this velocity until stop signal is issued <p>[car[m]=moving] during [tt(car[m]=moving), StopSignal] implies [speed[m]'=2mpss init speed[m]=0mps] during [tt(car[m]=moving), tr(speed[m]=30mps)] then [speed[m]'=0mpss init speed[m]=30mps] during [tr(speed[m]=30mps), StopSignal]</p>

3 CSL Grammar

Below CSL grammar for first version implementation is presented in BNF notation. Not all constructs described above are intended for implementation in the first version. Also, the definition of context related terms (free values and related intervals), and structured variables are still missing.

In the BNF specification, we use the conventions: $\{def\}^*$ to denote 0 or more repetitions def , and $\{def\}^+$ to denote 1 or more repetitions of def . Also, *names* are in *italic*, and we assume infinite sets of names, as follows:

- Blocks' names with typical representative: *id*.
- pwc- variables' names with typical representatives: x, y for variables in general, d for discrete variables, and b for Boolean variables'.
- Values' names with typical representatives: $v, v1, v2$.
- Events' names with typical representatives: e for events in general, and e_v for valued events. Also, n, m will serve as typical representatives for the Natural numbers, and r, q for Real numbers

```
<HRC-Specification> ::= HRC <Id>
                        Interface
                        Input: "{"<variables-definition>"}"
                        Output: "{"<variables-definition>"}"
                        Internal: "{"<variables-definition>"}"
                        Contracts "{"<contracts-specification>"}"
```

```
<variables-definition> ::= { <pwc-variable-definition> | <event-definition> }
```

```
<pwc-variable-definition> ::=  $x$  : <domain>
```

```
<domain > ::= <continuous-domain> | <discrete-domain>
```

```
<discrete-domain> ::= "{"  $v1, v2$  { $v$ } "
```

```
<condition> ::=  $b$  | <derived-condition>
                | <condition>"|"<condition> | <condition>"&"<condition>
                | "~"<condition> | if <condition> then <condition>
                | <condition> iff <condition>
```

```
<derived-condition> :=  $x$  <rel>  $v$  |  $x$  <rel> <pwc-function> |  $x =$  <timer-exp>
```

```
<rel> ::= < | = | > | <= | >=
```

```
<pwc-function> :=  $x$  | pwc( $e_v$ ) init  $v$  | prev( $d$ ) |  $F$  init  $v$  -- F is a continuous function
```

```
<timer-exp> := Timer(<T>) at  $e$  | PeriodicTimer(<T>) at  $e$ 
```

```
<T> ::=  $n$  time-unit4
```

⁴ **time-unit** should be replaced by the application basic time unit (second, mls, etc.). Recall, the physical interpretation of **time-unit** shall be assigned by application. Additional time units can be defined either by derived events.

<event-definition> ::= **event** e [:<discrete-domain>] | **event** e **defined-by** <state-change>

<state-change> ::= **event**(x) | **tr**(<condition>) | **fs**(<condition>) | e +<T>
| **periodic**(e , <T>) | **periodic**(<T>) |

<event> ::= e | e - v <rel> v | **startup**

<event-exp> ::= <event> | <event-exp> **when** <condition> | <event-exp> "|" <event-exp>
| <event-exp> "&" <event-exp> | <event-exp> "-" <event-exp>
| <event-exp> ";" <event-exp> | n "#" <event-exp>

<interval> ::= <lb> <event-exp> {,<event-exp>} <rb> | <lb><condition><rb>
| <lb> n : <event-exp> <rb> | <lb> T <rb>

<lb> ::= [| (

<rb> ::=] |)

<contract-specification> ::= {<viewpoint-id>} **contract** {id}
Assumption: {<assertion>}
Promise: {<assertion>}

<viewpoint-id> ::= **functionality/performance/timing/safety** -- other can be added.

<assertion> ::= <pattern> {, <pattern> }

<pattern> ::=

whenever "["<E>"]" **occurs** "["<C>"]" **holds during following** "["<I>"]"

| **whenever** "["<E>"]" **occurs** "["<E>"]" **implies** "["<E>"]" **during following** "["<I>"]"

| **whenever** "["<E>"]" **occurs** "["<E>"]" **does not occur during following** "["<I>"]"

| **whenever** "["<E>"]" **occurs** "["<S>"]" **occurs within** "["<I>"]"

| "["<C>"]" **during** "["<I>"]" **raises** "["<E>"]"

| "["<E>"]" **occurs** "["<N>"]" **times during** "["<I>"]" **raises** "["<E>"]"

| "["<E>"]" **occurs at most** "["<N>"]" **times during** "["<I>"]"

| "["<C>"]" **during** "["<I>"]"

implies "["<C>"]" **during** "["<I>"]" **then** "["<C>"]" **during** "["<I>"]"

<E> ::= <event-exp>

<C> ::= <condition>

<S> ::= <event-exp> | <condition>

<I> ::= <interval>

<N> ::= Natural