TGHC: Timed Guarded Horn Clauses

Kenji Saito
Department of Computer Science
Cornell University
Ithaca NY 14853 U.S.A.

Abstract

This paper describes the design principles, syntax, and semantics of the distributed real-time programming language TGHC *(Timed Guarded Horn Clauses).*

TGHC *is a descendant of concurrent logic programming languages [5, 6, 25, 28], and it is capable of explicitly expressing timing constraints by introducing the* timed guard *to* GHC *[28]. A formal semantics of a subset of* TGHC *is given as an appendix.*

1 Introduction

1.1 Motivations

Real-time systems [15] are systems whose timing behaviors adhere to the timing constraints imposed by the surrounding environments. Such systems should often be distributed since *a*) parallelism is demanded to achieve the required performance, *b*) the nature of the problems frequently requires components of the systems to be physically distributed (*e.g.* two robot arms pick a box up together), and *c*) redundant components are effective in order to tolerate failures (thus supporting the systems to meet their timing constraints).

Application fields of real-time systems have been expanding, and many such systems have been designed such as factory automation, avionics, and huge reactor system control. Due to the decrease of computer systems' cost and to the increase of their performance, there will be even wider applications of real-time systems in the future. The concept of *HFDS* (Highly Functionally Distributed Systems) in TRON project, as Sakamura showed in [20, 22], illustrates this picture of the future where virtually everything is computerized.

However, it is difficult to build huge real-time systems as *HFDS* by using today's technologies; the following is a list of some of the difficulties:

1. Such systems contain a large number of real-time programs that cooperate – productivity has to be improved, and concurrent programming has to be made easy.

2. Such systems necessitate explicit handling of time. Designers of such systems have to consider timing behaviors of the systems throughout their specifications, implementations, and verifications – methodologies for this have to be established.

3. Since such systems interact with human society, failures of such systems can harm the society to a considerable extent; they could even be life-threatening – fault-tolerance and automatic verification of the systems are required.

We believe that most of these problems can be made easier by providing appropriate programming languages.

In today's real world, real-time systems are programmed mostly in low-level languages such as *C* and assembly languages, for reasons mainly concerning efficiency in their execution; but efficiency in programming deserves much more attention. These languages lack capabilities in expressing timing constraints and concurrency, two most important properties of real-time programs; these have to be taken care of by capabilities of operating systems, namely scheduling policies and multi-tasking, respectively. As a result, these properties are not made very explicit in the programs themselves, making it troublesome to modify, verify, and reuse real-time programs.

A number of imperative real-time programming languages have been proposed to improve this situation; such languages include *Ada* [12] and real-time concurrent versions of *C* [9]. However, these languages are hereditarily sequential (two statements in a program are executed sequentially by default), and they are not suitable for abstraction of problems that are of concurrent nature. Also, these languages became more complex by incorporating more syntactic constructs to introduce timing and concurrency. Furthermore, due to side effects of assignments, behaviors of the written programs are difficult to reason about.

We believe that real-time programming languages in the future should rather be declarative than imperative. Reasoning about declarative programs is easier than reasoning about imperative programs, since semantics of declarative languages are based on static (mathematical) frameworks such as function, logic, and algebra. Also, concurrent programming with declarative languages is meant to be easier than imperative languages. Mathematical expressions do not contain control informations that specify how to evaluate (or prove) them; some parts of them can be evaluated (or proved) in an arbitrary order, and some parts of them depend on others. By describing given problems with mathematical expressions, programmers can naturally disclose concurrency hidden in the problems.

1063-6749/92 $3.00 ©1992 IEEE

Moreover, these languages would provide means to specify timing relations between events, which is easier than specifying procedures to meet timing constraints.

The goal of this paper is to propose a new distributed real-time programming language *TGHC* (Timed Guarded Horn Clauses), a descendant of concurrent logic programming languages [5, 6, 25, 28], which provides explicit notions of timing and concurrency in a simple and declarative way.

1.2 Approach and related works

Related works *Real-Time Lucid* [8], *RLucid* [18], and *LUSTRE* [3] are declarative real-time languages. These languages are based on operations over timed streams, and are categorized as data-flow languages. *RLucid* and *LUSTRE* are based on *multiform time* point of view; any streams of external events (such as values sent from sensors) are viewed as defining clocks, and actions are taken at the exactly same time as the clocks tick – a clock ticks when an event arrives (therefore *LUSTRE* is often regarded as a *synchronous real-time language*). Programs based on multiform time do not need to handle time values, and are claimed to be more reusable, since they can work on different time scales. These declarative real-time languages are also intended to be used as specification languages. Halbwachs et al. showed in [11] a way to verify a real-time program by using *LUSTRE* as both its specification and implementation language.

Some languages have been designed in order to construct reactive environments with existing real-time programs; the act of constructing reactive environments is to program (or specify) interfaces of the environments, and therefore can be called *meta-programming*. Such languages include *NPL* [31] (based on guarded commands) and *TACL/TULS* [14, 21] (based on macro expansion). *NPL* introduces a notion of time to guarded commands by placing timers in their guards. *TACL/TULS*, as far as we could tell, does not seem to have any notions of time. These languages have relatively simple semantics since they are not intended for complex programming, so that mathematical semantics can be formulated. However, behaviors of the meta-programs depend on the behaviors of the controlled real-time programs, which are rather unpredictable.

Approach in TGHC Concurrent logic programming languages are based on guarded commands, as well as Horn clauses. *TGHC* introduces a notion of time by placing time intervals after certain parts of guards (called *timed guards*) become true. Since programming in concurrent logic programming languages is conventionally based on stream operations, this makes timed streams as used in *LUSTRE* available in *TGHC*. *LUSTRE* gives higher abstractions of timed streams, while *TGHC* gives more control over timed streams; whether providing more control over timed streams is beneficial in real-time programming or not is yet to be studied, since it makes efficient implementation more difficult.

Since meta-programming [1] has been studied extensively in concurrent logic programming [7, 19, 26, 28], the results may apply to *TGHC*. A meta-interpreter of *TGHC* is shown later in Section 4.3.

1.3 Structure of this paper

The rest of this paper is organized as follows: Section 2 gives explanation of necessary background to understand this paper. Section 3 describes the design principles, syntax, and semantics of *TGHC*. Section 4 shows some examples of *TGHC* programs. Finally, section 5 gives concluding remarks. An appendix gives a formal semantics of a subset of *TGHC*.

2 Background

2.1 Distributed real-time programs

Outline Distributed real-time programs are concurrent programs, whose processes are phyically distributed over networks, and are timed in relative to either internal or external events. An example of a process timed in relative to internal events is a periodic process that samples the value of a sensor once in a specific time interval. An example of a process timed in relative to external events is a sporadic process that has to take an action within a specific time interval after a sampled value of a sensor exceeds certain range.

In most cases, it is more important to constrain reaction time of the programs to the events in the physical world than to constrain the whole length of their execution time; distributed real-time programs do not necessarily terminate, in which case there is no sense in constraining completion time of the programs (termination of flight control programs or nuclear reactor control programs is not desirable).

The fact that distributed real-time programs interact with the physical world implies that behaviors of these programs have indeterminacy; the behaviors of the programs cannot be decided only by looking at the programs themselves – they depend on the ordering and timing of the events in the physical world (this also applies to any concurrent programs; after all, the physical world can be abstracted as another process).

Timed stream A timed stream is a sequence of time-stamped values. Timed streams are useful abstraction to be used in continuing interactions among processes in distributed real-time programs. A process can take a timed stream as an input sequence of events, take necessary actions, and generate another timed stream as an output. In *LUSTRE*, a program is a set of nodes, which define functions from streams to streams. In *TGHC*, a program is a set of clauses, which define relations among streams.

[1] Meta-programming in concurrent logic programming is intended to add more functionality to the runtime systems, and is not necessarily concerned with constructing reactive environments. However, it gives detailed control over multiple programs, and can be applied to construction of reactive environments.

2.2 Concurrent logic programming languages

Outline Concurrent logic programming languages, such as *Concurrent Prolog* [24, 25], *PARLOG* [6], and *GHC* [28], are descendant of the *Relational Language* [5][2]; they are based on the notion that concurrent programs are relational than functional, because of their indeterminacy. Their syntaxes and declarative semantics are almost the same; they are all based on *guarded Horn clauses*, which introduces the commit operator to Horn clauses. But their operational semantics are rather different in their synchronization mechanisms.

In this subsection, informal syntax and semantics of *GHC* are described, since current semantics of *TGHC* is based on that of *GHC* (readers are referred to appendix A for formal definitions). An example of a *GHC* program is also described, which is extended to a *TGHC* program in section 4.1.

Preliminaries The basic elements of *GHC* programs are *terms*. A term is formed from function symbols and variables. Traditionally, function symbols are symbols beginning with small letters, and variables are those beginning with capital letters. A variable alone is a term. Otherwise, a term is of the following form:

$$f(\gamma_1, \gamma_2, \ldots, \gamma_n) \ (n \geq 0)$$

where f is a function symbol, and γ's are terms. For example, X, a, $cons(a, X)$, $cons(a, cons(b, X))$ are terms.

An *atom(ic formula)* is of the following form:

$$p(\gamma_1, \gamma_2, \ldots, \gamma_n) \ (n \geq 0)$$

where p is a predicate symbol, and γ's are terms. Traditionally, predicate symbols are symbols beginning with small letters. For example, $is\_list(cons(a, X))$ is an atom.

Syntax of GHC A *GHC* program is a set of guarded Horn clauses of the following form:

$$H \leftarrow G_1, \ldots, G_m \mid B_1, \ldots, B_n. \ (m, n \geq 0)$$

where H, G's, and B's are atoms. H is called the *head*, and G's and B's are called *guard goals* and *body goals*, respectively. A goal may be a *unification goal* of the following form:

$$\gamma_1 = \gamma_2$$

where γ_1 and γ_2 are terms, or a non-unification goal that is an atom. "|" is the *commit operator*. H and G's together are called the *guard*, and B's as a group are often called the *body*. If there are no goals in the guard or body, it is traditionally denoted by *true*.

A program is invoked by a *goal clause* of the following form:

$$\leftarrow B_1, \ldots, B_n. \ (n \geq 0)$$

where B's are goals.

[2]Papers referenced here are all found in the collected papers edited by Shapiro [27].

Declarative semantics of GHC A guarded Horn clause is read as follows:

If every goal in its guard and body is true, its head is true.

Results of every successful[3] execution of *GHC* programs conform the above reading of applied guarded Horn clauses; this is called *soundness*. There may be some clauses that are not applied in an execution, so that the results might not be the only solution; this is called *incompleteness*. Incompleteness reflects the fact that there is indeterminacy in concurrent programs.

Operational semantics of GHC Intuitively, each guarded Horn clause is considered as a rewrite rule of a goal, where its guard specifies the conditions to be satisfied for the rule to be applied, and its body specifies the actual goals to replace with. If more than one clause can be applied, one of them is selected nondeterministically. The act of applying a clause is called *commitment*. Commitment is an irreversible act.

Assignment of variables is usually called *binding*. Bindings are produced after commitments, and any attempts to bind the bound variables with incompatible terms fail.

Two terms are *unified* when they become lexically identical by binding the variables in each with the corresponding terms of the other, and replacing the variables with the terms.

A goal is *instantiated* when any variables appearing in its arguments are bound.

An informal operational semantics of *GHC* is given as follows:

1. Goal execution: Every goal in the goal clause is executed concurrently; a goal is executed by the following steps:
   - (a) Head unification: Variables appearing in the head of a clause is analogous with formal parameters of procedural or functional programming languages. Clauses whose heads are unifiable with the calling goal become the candidates for commitment; variables appearing in their heads are bound with the corresponding terms in the calling goal, and replaced by them.
   - (b) Suspension rule: Guard goals of the candidates are executed concurrently, with a restriction imposed by the suspension rule: any attempts to instantiate the calling goal are suspended.
   - (c) Commitment: The execution of the calling goal commits to a clause whose guard succeeds; the body of the committed clause replaces the calling goal. Unification goals in the body may instantiate the calling goal.

[3]Maher investigates in [16] the conditions when a failed execution is sound – meaning there is no solution to the given goals.

2. Success: A unification goal succeeds if its arguments are unified; a non-unification goal succeeds if it is eventually replaced by unification goals that succeed, or by an empty body. A guard succeeds if every guard goal succeeds. A program succeeds if every goal in its goal clause succeeds.

3. Failure: A unification goal fails if their arguments are not unifiable; a non-unification goal fails if its execution has no candidates for commitment, or the guard of every candidate fails. A guard fails if any of its guard goals fail. A program fails if any goals in its goal clause fail.

Process interpretation of GHC programs A *GHC* program defines a concurrent program in the following way:

1. Recursively defined predicates define processes.
2. Conjunction of processes define a network of processes.
3. Arguments of the goals define local states of processes.
4. Shared variables among goals define communication channels.

An example of a *GHC* program is described below.

Example: A railway control problem The following problem is taken from [11]:

> Figure 1 shows a railroad with two embedded sensors, *Sensor*1 and *Sensor*2, which return *true* whenever there are wheels on them, and return *false* if not. The problem is to detect the traversal of each axle from one district to the other. There can be only one axle in *Z* zone at a time, but an axle may turn back within *Z* zone, and even oscillate on or about *Z* zone.

The above problem can be solved by defining three processes: two sampling processes to sample the values of *Sensor*1 and *Sensor*2, and a detecting process to monitor rising and falling of sampled values to detect the traversal of each axle. It often helps to draw state transition diagrams before writing a *GHC* program. Figure 2 shows state transition diagrams of this problem, which is read as follows:

A sampling process (a) has only one state; it keeps sampling the value of the sensor to which the process is assigned. The detecting process (b) has three states: *ST*0 (initial state), *ST*1 (after an axle entered *Z* zone from the left district), and *ST*2 (after an axle entered *Z* zone from the right district). While the process is in *ST*0,

1. An axle enters *Z* zone from left when *Sensor*1 rises while *Sensor*2 is *false*.
2. An axle enters *Z* zone from right when *Sensor*2 rises while *Sensor*1 is *false*.

While the process is in *ST*1 (*ST*2),

1. An axle traverses from left to right (right to left) when *Sensor*2 (*Sensor*1) falls while *Sensor*1 (*Sensor*2) is *false*.
2. An axle turns back when *Sensor*1 (*Sensor*2) falls while *Sensor*2 (*Sensor*1) is *false*.

Otherwise the process stays in the same state.

Figure 3 shows a *GHC* program translated from the state transition diagrams.

There are some conventions about *GHC* program text: A text following % is a comment. $[a|b]$, $[a,b|c]$, and [ ] are syntactic sugar for $cons(a,b)$, $cons(a, cons(b,c))$, and *nil*, respectively (although [ ] does not appear in the example). An anonymous variable, denoted by _, is used when any terms may be bound to the variable.

:= is a built-in binary predicate that unifies the left argument with the evaluation of the right argument. *otherwise* is a built-in predicate that can only appear in guards; it succeeds when the guard of every other candidate clause fails.

In Figure 3, we introduced two built-in functions *sense*1() and *sense*2(), which have the values of *Sensor*1 and *Sensor*2, respectively, when evaluated.

In *GHC*, a stream is represented by a list. Process *sensor*1 and *sensor*2 in Figure 3 are typical producers of streams; their arguments are output streams created by the processes – the unification goal $S = [X|S']$ adds an element to the stream.

Process *detector* in Figure 3 is a typical consumer of streams; its first argument is its state, second argument is the input stream from process *sensor*1, third argument is the input stream from process *sensor*2, and the fourth argument is the output stream. A term *lToR* is added to the output stream when an axle traverses from left to right, and *rToL* is added when an axle traverses from right to left.

In this solution, there must be no axles on either *Sensor*1 or *Sensor*2 when the program starts.

As readers may have noticed, this program probably won't work. Since the operational semantics of *GHC* does not specify the relative speed among processes, one of the sampling processes may run faster than the other; this means that the decision of the detecting process is based on values of *Sensor*1 and *Sensor*2 that are sampled at two different points of time. In this case the detection is not at all reliable. Even if the two sampling processes sampled the values at exactly the same rate, the values are of no use if these are not sampled frequently enough in relative to the speed of axles. Furthermore, even if the two sampling processes sampled the values at exactly the same rate and frequently enough, detection does not mean a thing if the detecting process monitored sampled values infrequently, say, once in two hours.

The above problems are solved by a *TGHC* version of the program described in section 4.1.

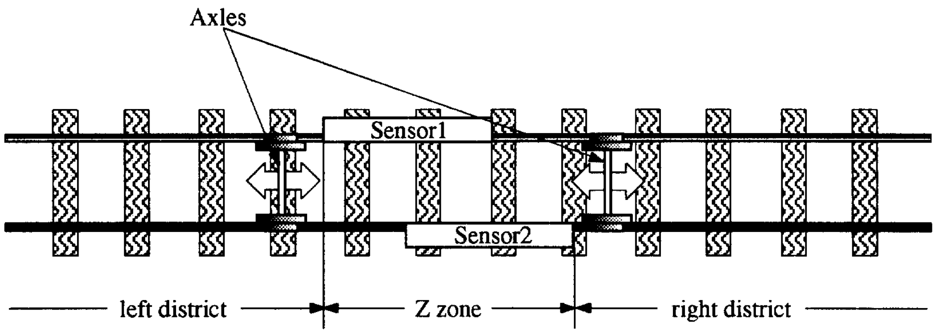

Figure 1: A railway control problem

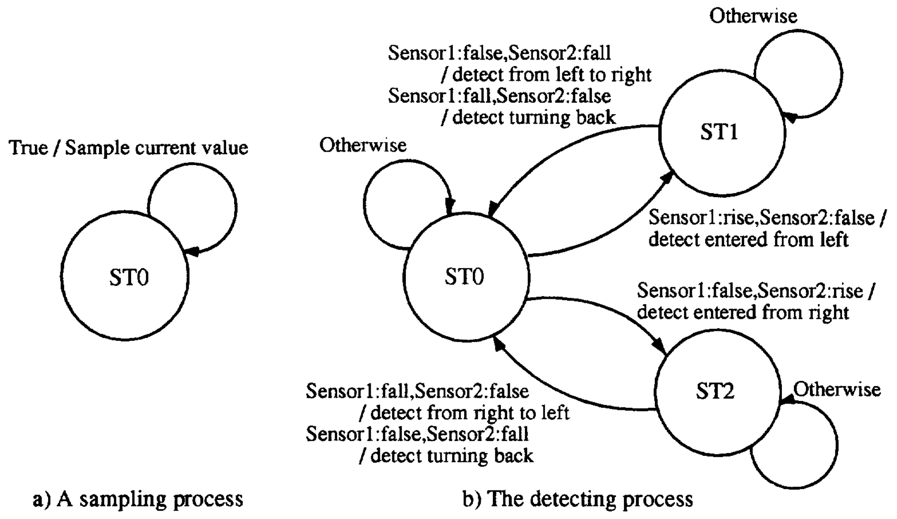
True / Sample current value
ST0
a) A sampling process
Otherwise
Sensor1:false,Sensor2:fall
/ detect from left to right
Sensor1:fall,Sensor2:false
/ detect turning back
Otherwise
ST1
ST0
Sensor1:rise,Sensor2:false /
detect entered from left
Sensor1:false,Sensor2:rise /
detect entered from right
ST2
Otherwise
Sensor1:fall,Sensor2:false
/ detect from right to left
Sensor1:false,Sensor2:fall
/ detect turning back
b) The detecting process

Figure 2: State transition diagrams of the railway control problem

% Process *sensor1* samples the value of *Sensor1*.
$sensor1(S) \leftarrow true \mid X := sense1(), S = [X|S'], sensor1(S')$.

% Process *sensor2* samples the value of *Sensor2*.
$sensor2(S) \leftarrow true \mid X := sense2(), S = [X|S'], sensor2(S')$.

% Process *detector* detects the traversal of each axle.
% *ST0* state
$detector(st0, [false, true|S1], [false, false|S2], D) \leftarrow true \mid detector(st1, [true|S1], [false|S2], D)$.
$detector(st0, [false, false|S1], [false, true|S2], D) \leftarrow true \mid detector(st2, [false|S1], [true|S2], D)$.
% *ST1* state
$detector(st1, [false, false|S1], [true, false|S2], D) \leftarrow true \mid D = [lToR|D'],$
    $detector(st0, [false|S1], [false|S2], D')$.
$detector(st1, [true, false|S1], [false, false|S2], D) \leftarrow true \mid detector(st0, [false|S1], [false|S2], D)$.
% *ST2* state
$detector(st2, [true, false|S1], [false, false|S2], D) \leftarrow true \mid D = [rToL|D'],$
    $detector(st0, [false|S1], [false|S2], D')$.
$detector(st2, [false, false|S1], [true, false|S2], D) \leftarrow true \mid detector(st0, [false|S1], [false|S2], D)$.
% When state does not change
$detector(ST, [\_, V1|S1], [\_, V2|S2], D) \leftarrow otherwise \mid detector(ST, [V1|S1], [V2|S2], D)$.

% Goal clause
$\leftarrow sensor1(S1), sensor2(S2), detector(st0, [false|S1], [false|S2], D)$.

Figure 3: An example of a *GHC* program

3 Timed Guarded Horn Clauses

3.1 Design principles

Separation of concurrency and parallelism In *GHC/KL1* approach [4, 30], *GHC* is only concerned with concurrency, and *KL1*, a practical version of *GHC*, is responsible for parallelism[4]. This benefits the development of concurrent programs in the following way:

1. In specification stage, a specification can be written as an executable *GHC* program.
2. In implementation stage, the executable *GHC* program can be restricted to an equivalent *KL1* program to fit the physical setting.
3. A way to verify the implementation is to prove that the *GHC* program can be transformed to the *KL1* program by using appropriate transformation rules.

TGHC should follow this approach; there will be the abstract version and the practical version of *TGHC*. This paper only describes the abstract version.

Separation of timing specification and meeting the specification In *GHC/KL1* approach, however, timing behaviors of programs are considered as a part of parallelism; this is not always helpful, since many real problems naturally involve timing (*e.g.* the railway control problem we saw in section 2.2), and *GHC* programs cannot be the specifications of their solutions. In *TGHC*, Timing should be specified by programs written in the abstract version; how to meet the specification should be worried by the practical version. In this way, the methods of implementing and verifying programs described in the paragraph before can also be applied to real-time programs.

The following subsections give informal syntax and semantics of *TGHC* (readers are referred to appendix A for formal definitions).

3.2 Syntax

A *TGHC* program is a set of *timed guarded Horn clauses*; a timed guarded Horn clause divides the guard goals of a guarded Horn clause into two groups as follows:

$$H \leftarrow G_1, \ldots, G_m :: G_{t1}, \ldots, G_{tk}\ [l, u]$$
$$\mid B_1, \ldots, B_n.\ (m, n, k \geq 0)$$

where G_t's as a group are called the *timed guard*. l and u are non-negative integers including ∞, and it denotes a time interval; l is the lower bound of the time interval, and u is the upper bound of the time interval. The time interval $[0, \infty]$ can be omitted.

3.3 Declarative semantics

The declarative semantics of a timed guarded Horn clause is the same as that of its equivalent guarded Horn clause obtained by ignoring "::" and $[l, u]$.

[4]How concurrent programs are executed. For example, physical distribution of processes is a part of parallelism.

The intended timing behavior of a timed guarded Horn clause is that every successful execution of *TGHC* programs guarantees that no clauses are committed either before the lower bound of their time intervals or after the upper bound of their time intervals, measured from the time point when its timed guard succeeds. In other words, a timed guarded Horn clause is read as follows:

> ***If every goal in its guard and body is true, its head is true, but the clause is only applicable within its time interval after every goal in its timed guard becomes true.***

The mapping between the time unit used in the time intervals and the physical time may depend on each program. *TGHC* assumes a global time within a program.

3.4 Operational semantics

The operational semantics of a *TGHC* program is the same as that of its equivalent *GHC* program except the following:

Timed commitment The unification goals in the body of a clause either succeed or fail at the time of the commitment (otherwise they must be suspended due to the suspension rule; this would only happen if the clause is called by a goal in the guard of some clause).

An execution can commit to a clause whose both untimed and timed guard succeed; but when the execution can commit is constrained by the time when the timed guard succeeds.

An execution must commit to a clause that has the lower bound l and the upper bound u of the time interval at time t such that $t_g + l \leq t \leq t_g + u$, where t_g is the time when the timed guard of the clause succeeds. A timed guard succeeds when every goal in it succeeds. The time when a goal succeeds is defined as follows:

1. A unification goal succeeds at the time of the latest binding that is necessary to make the arguments of the goal unifiable without suspension.

2. Other goals succeed at the time when every goal in their committed clauses succeed.

If the timed guard of a clause is empty (or a tautology), the time interval denotes the interval measured from the beginning of the execution of the program.

Timed evaluation of functions A physical value (such as temparature, pressure, etc.) is represented by an evaluation of a function. If the body of a clause contains evaluations of functions (such as ones used by a built-in predicate :=), the functions are evaluated at the time of the commitment.

% Predicate *bound* succeeds when a value is sampled.
$bound(X) \leftarrow true :: X = true\ [0,0] \mid true.$
$bound(X) \leftarrow true :: X = false\ [0,0] \mid true.$

Figure 6: Predicate *bound* for the railway control program

% Process *detector* now detects time out, and raises
% timing exception after 5 time units the previous
% values of sensors are sampled.
$detector(ST, [V1|S1], [V2|S2], D) \leftarrow true ::$
$\quad bound(V1), bound(V2)\ [5,5] \mid$
$\quad D = [exc(timing)|D'],$
$\quad detector(ST, S1, S2, D').$

Figure 7: Timing exception for process *detector* in the railway control program

4 Example Programs

4.1 Example A: Railway control program

Now let us review what was wrong with the *GHC* program in Figure 3. Apparently, the program ignored timing constraints that are implied in the solution. First, we make the timing constraints explicit by drawing *timed state transition diagrams.*

Figure 4 shows timed state transition diagrams of the railway control program. In the figure, $[l, u]$ denotes the time interval of a transition. The diagrams are based on the assumption that the appropriate interval for sampling values of sensors is defined by one time unit, and the maximum delay of detection is three times longer than the interval. Note that the state transition of a sampling process is timed in relative to its internal event, namely sampling the previous value.

Now it is a straightforward translation from the diagrams to a *TGHC* program. Figure 5 is an example of such a program; it is also an extension of the *GHC* program in Figure 3.

In Figure 5, process *sensor*1 and *sensor*2 are extended to accomplish periodic execution. They receive the previous values they sampled as a part of their arguments. A built-in predicate *bound* becomes true when its argument is a non-variable term, and the current values have to be sampled exactly one time unit after the previous values are sampled.

In Figure 5, process *detector* is extended to meet its deadline. It checks the current values of the sensors in the timed guards of its clauses, making its state transition occur within three time units after these values are sampled; as a result, the traversal is detected within the time limit.

Predicate *bound* in Figure 5 can also be implemented by a *TGHC* program as shown in Figure 6.

It is natural to ask what would happen if the timing specification is not met. According to the operational semantics of *TGHC*, the execution of the program fails. However, it is not always desirable. We

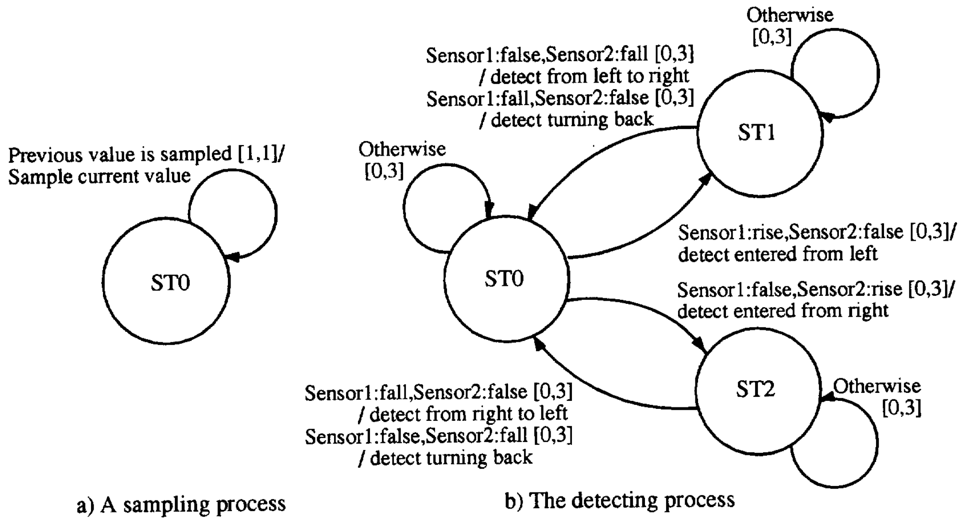
Previous value is sampled [1,1]/
Sample current value
ST0
a) A sampling process
Otherwise
[0,3]
Sensor1:false,Sensor2:fall [0,3]
/ detect from left to right
Sensor1:fall,Sensor2:false [0,3]
/ detect turning back
ST1
Otherwise
[0,3]
ST0
Sensor1:rise,Sensor2:false [0,3]/
detect entered from left
Sensor1:false,Sensor2:rise [0,3]/
detect entered from right
ST2
Otherwise
[0,3]
Sensor1:fall,Sensor2:false [0,3]
/ detect from right to left
Sensor1:false,Sensor2:fall [0,3]
/ detect turning back
b) The detecting process

Figure 4: Timed state transition diagrams of the railway control program

% Process *sensor1* samples the value of *Sensor1* once per 1 time unit.
$sensor1([P|S]) \leftarrow true :: bound(P)\ [1,1]\ |\ X := sense1(), S = [X|S'], sensor1(S).$

% Process *sensor2* samples the value of *Sensor2* once per 1 time unit.
$sensor2([P|S]) \leftarrow true :: bound(P)\ [1,1]\ |\ X := sense2(), S = [X|S'], sensor2(S).$

% Process *detector* detects the traversal of each axle within 3 time units.
% *ST0* state
$detector(st0, [false, V1|S1], [false, V2|S2], D) \leftarrow true :: V1 = true, V2 = false\ [0,3]\ |$
$\quad detector(st1, [V|S1], [V|S2], D).$
$detector(st0, [false, V1|S1], [false, V2|S2], D) \leftarrow true :: V1 = false, V2 = true\ [0,3]\ |$
$\quad detector(st2, [V1|S1], [V2|S2], D).$
% *ST1* state
$detector(st1, [false, V1|S1], [true, V2|S2], D) \leftarrow true :: V1 = false, V2 = false\ [0,3]\ |$
$\quad D = [lToR|D'], detector(st0, [V1|S1], [V2|S2], D').$
$detector(st1, [true, V1|S1], [false, V2|S2], D) \leftarrow true :: V1 = false, V2 = false\ [0,3]\ |$
$\quad detector(st0, [V1|S1], [V2|S2], D).$
% *ST2* state
$detector(st2, [true, V1|S1], [false, V2|S2], D) \leftarrow true :: V1 = false, V2 = false\ [0,3]\ |$
$\quad D = [rToL|D'], detector(st0, [V1|S1], [V2|S2], D').$
$detector(st2, [false, V1|S1], [true, V2|S2], D) \leftarrow true :: V1 = false, V2 = false\ [0,3]\ |$
$\quad detector(st0, [V1|S1], [V2|S2], D).$
% When the state does not change
$detector(ST, [\_, V1|S1], [\_, V2|S2], D) \leftarrow otherwise :: bound(V1), bound(V2)\ [0,3]\ |$
$\quad detector(ST, [V1|S1], [V2|S2], D).$

% Goal clause
$\leftarrow sensor1(S1), sensor2(S2), detector(st0, S1, S2, D), S1 = [false|S1'], S2 = [false|S2'].$

Figure 5: Example A: railway control program

```
% Process merge merges two timed streams
% in the order of appearance.
merge([A|X], Y, Z) ← true :: bound(A)[0,0]
      | Z = [A|Z'], merge(X, Y, Z').
merge(X, [A|Y], Z) ← true :: bound(A)[0,0]
      | Z = [A|Z'], merge(X, Y, Z').
```

Figure 8: Example B: Binary merge of timed streams

should expect ourselves to be strongly encouraged to make our railway control progam non-stop for the benefit of passengers.

As an example of dealing with timing errors, now we make process *detector* to detect its time out as well, by adding a clause shown in Figure 7 to the program; after five time units the previous values of sensors are sampled, process *detector* raises timing exception by adding a term *exc(timing)* to its output stream.

Note that three kinds of timing constraints, periodic execution, deadline, and time out, are all accomplished by a single concept of timed commitment in *TGHC*.

A different solution of the railway control problem written in *LUSTRE* is found in [11]. Readers may want to compare the two solutions.

4.2 Example B: Binary merge of timed streams

The railway control program described in section 4.1 is itself a high-level sensor that senses the traversal of each axle. The whole railway control system would consist of many such high-level sensors, high-level control programs, and high-level actuators. The high-level control programs would be simplified if we could merge the output streams of high-level sensors in the order of occurrence of events, and construct a single input stream for the control program.

Figure 8 shows a process that merges two streams in the order of occurrence of bindings. The first and the second arguments of the process is the two input streams, and the third argument is the output stream. An element of the input streams is added to the output stream as soon as it is bound, preserving the order of bindings.

This program is also a straightforward extension of binary merge program in *GHC*, found for example in [28].

4.3 Example C: Meta-interpreter

We saw in section 4.1 that we can make process *detector* to detect time out and to raise timing exception by adding a clause. However, adding this functionality to each process in the whole railway control system may result in loss of productivity and reliability, since in this way every consumer of streams have to deal with the timing exception of the producers, forcing programmers to write more code, and making processes less applicable to other systems.

We may want to deal with timing exceptions in meta-level, by extending the runtime system of *TGHC* to detect and handle timing exceptions. A way

```
% Process call evaluates goals.
call(true, Where) ← true :: true [0,0] | true.
call((A, B), Where) ← true :: true
      | call(A, Where), call(B, Where).
call(X, guard) ← X = (A = B) ::
      A = B [0,0] | true.
call(X, body) ← true :: X = (A = B) [0,0]
      | A = B.
call(A, Where) ← A \= true, A \= (_, _),
      A \= (_ = _) :: true |
      clauses(A, Clauses),
      resolve(A, Clauses, Body),
      call(Body, body).

% Process resolve tries the candidates.
resolve(A, [C|Cs], B) ←
      melt_new(C, (A ← G :: Gt [L, U] | B')),
      call(G, guard) :: call(Gt, guard) [L, U] | B = B'.
resolve(A, [C|Cs], B) ← true ::
      resolve(A, Cs, B') [0,0] | B = B'.
```

Figure 9: Example C: Meta-interpreter

to do this is to write an interpreter of *TGHC* in *TGHC* itself; this kind of interpreters are called *meta-interpreters*.

Figure 9 shows an experimental meta-interpreter of *TGHC*, obtained by extending the meta-interpreter of *GHC* in [28]. This interpreter does not have extra functionality, and such interpreters are often called *vanilla* meta-interpreters.

An execution of a program through this meta-interpreter is invoked by a goal clause such as follows:

$$\leftarrow call((p1(X), p2(X,Y), p3(Y)), body).$$

Process *call* in Figure 9 executes goals, and process *resolve* accomplish commitments.

Calling goals from the guard and from the body are distinguished by the second parameter of process *call*; this is essential in dealing with unification goals. Calling unification goals in the guard must succeed when the unification goals succeed, whereas calling unification goals in the body must try to unify the arguments of the goal as soon as the clause is committed.

A built-in predicate *clauses* outputs a stream as its second argument, which contains clauses whose head is unifiable with the first argument. A built-in predicate *melt_new* does the head unification, and build a new executable clause with new local variables (otherwise every local variable in the clause may not be bound in the guard according to the suspension rule, as explained in [28]).

By extending this meta-interpreter, we can implement detection and handling of timing error in programs in a uniform manner.

The problem with this meta-interpreter is that it cannot reflect on failure; the meta-interpreter will fail if the program that is executed through the meta-interpreter fails. This is undesirable, since it means that failure in one part of a system is inevitably propagated to the whole system. We may have to enhance the semantics of *TGHC* to solve this problem, namely introducing *tell guard*[13, 23] that enables reflection on failure.

5 Conclusion

5.1 Summary

A new distributed real-time programming language *TGHC* is proposed. *TGHC* is a decendant of concurrent logic programming languages, and its programs specify relations among timed streams. *TGHC* introduces notions of the timed guard and timed commitment to its direct ancestor, *GHC*. Three typical timing constraints, periodic execution, deadline, and time out, are all accomplished by the single concept of timed commitment. Some examples of *TGHC* programs are described.

TGHC is designed based on our belief that distributed real-time programming languages of the future should be declarative rather than imperative, in order to make writing and reasoning about both concurrent and timed programs easier. *LUSTRE* is an example of declarative real-time programming languages, yet it is based on functions, and thus cannot express indeterminacy in concurrent programs. *TGHC* is based on relations, and is able to express indeterminacy.

However, our research on *TGHC* has just begun, and there remains a lot to be studied. The following subsection discusses future works.

5.2 Future works

Predictable implementation The main concern in implementations of *TGHC* is the predictability of timing behaviors. The language features that make the programs' timing behaviors unpredictable should be restricted so that they can only be used in a way such that the timing behaviors remain predictable.

Handling unbounded recursive data structures within the guard of a clause should be restricted in some way, since it makes the execution time of programs unpredictable. A type system will be useful to allow implementations to represent a term internally in a way such that it can be accessed in a bound time. It also decreases programmers' mistakes.

Desirable operating system capabilities The other issue concerning implementation is capabilities of operating systems. Difficulties in implementing *TGHC* runtime systems should sometimes be overcome by enhancing capabilities of operating systems. We should clarify the class of such capabilities through implementing *TGHC* runtime systems. The following is a list of examples of such capabilities:

1. Time-based scheduling – each process in a *TGHC* program has its time interval; it is desirable for each process to be scheduled according to its period or deadline.
2. Group communication – processes in a *TGHC* program communicate each other via timed streams; a timed stream is always written by one process, and sometimes read by more than one process. This is naturally implemented by broadcast. It is desirable that operating system interfaces provide broadcast as one of their communication primitives.

Verification method Verification is extremely important for distributed real-time programs. Because of the indeterminacy, testing of such programs require generation of every possible behavior of the implementation; this often is infeasible, therefore testing may be imperfect. We need proofs that such programs do not exhibit wrong behaviors.

There are two major ways of verification: One is to specify the specification as formulas in real-time logics [1], and to prove that these formulas are valid on the model that represents the implementation (this method is explained in [2, 17]). The other is to write the specification and the implementation in the same programming language, and to prove that these two programs are equivalent by certain criteria (*e.g.* by transforming one program to another).

TGHC should be able to support whichever method is convenient for system designers. For the latter method, applicability of *GHC* program transformation rules [29] to *TGHC* programs should be studied.

Acknowledgement

We would like to thank Bard Bloom and Fred B. Schneider for comments on earlier drafts of this paper, and Takashi Chikayama and Kazunori Ueda for answering our questions reguarding *GHC/KL1*.

Appendix

A Formal Semantics of a subset of TGHC

This appendix defines a formal semantics of a subset of *TGHC* named *Flat TGHC*. The semantics uses *timed transition system* as its model.

A.1 Timed Transition System for Flat TGHC

Timed Transition System [10] is a model for specifying real-time systems; a timed transition system is a labeled transition system whose labels include time intervals for transitions. In this subsection, *Timed Transition System for Flat TGHC*, a variant tuned for the purpose of formulating the semantics of *Flat TGHC*, is described.

Transition System for Flat GHC, the untimed version of the transition system, is described first; then time is incorporated with the model.

Symbols We fix the infinite set $\mathcal{F}$ of function symbols, $\mathcal{P}$ of predicate symbols, and $\mathcal{V}$ of variable symbols.

Term Let Γ be the infinite set of *terms* such that:

1. $\mathcal{V} \subset \Gamma$.

2. For every $f \in \mathcal{F}$, where f is a n-ary function symbol ($n \geq 0$), $f(\gamma_1, \ldots, \gamma_n) \in \Gamma$ iff for every $1 \leq i \leq n$, $\gamma_i \in \Gamma$.

A congruence relation $s \subseteq \Gamma^2$ is a relation that satisfies the following conditions:

1. s is an equivalence relation (*i.e.* s is reflexive, symmetric, and transitive).

2. For every $f(\gamma_1, \ldots, \gamma_n), f(\gamma'_1, \ldots, \gamma'_n) \in \Gamma$ ($n \geq 0$), $f(\gamma'_1, \ldots, \gamma'_n) \in s(f(\gamma_1, \ldots, \gamma_n))$ iff for every $1 \leq i \leq n$, $\gamma'_i \in s(\gamma_i)$.

Atom Let $\mathcal{A}$ be the infinite set of *atoms* such that for every $p \in \mathcal{P}$, where p is a n-ary predicate symbol ($n \geq 0$), $p(\gamma_1, \ldots, \gamma_n) \in \mathcal{A}$ iff for every $1 \leq i \leq n$, $\gamma_i \in \Gamma$.

Transition system for Flat GHC A transition system Ψ is a tuple $< \mathcal{S}, \Theta, \mathcal{B}, \Lambda, \rightharpoonup >$, such that:

1. $\mathcal{S}$ is a partially ordered set of congruence relation $s \subseteq \Gamma^2$, such that for every $s, s' \in \mathcal{S}$, $s \preceq s'$ iff for every $\gamma \in \Gamma$, $s(\gamma) \subseteq s'(\gamma)$.

   For every $\gamma, \gamma' \in \Gamma$, γ and γ' are *not congruent* in $s \in \mathcal{S}$ iff the following conditions hold:

   (a) There are some $f(...) \in s(\gamma)$ and $f'(...) \in s(\gamma')$ such that $f \neq f'$, or

   (b) There are some $f(\gamma_1, \ldots, \gamma_n) \in s(\gamma)$ and $f(\gamma'_1, \ldots, \gamma'_n) \in s(\gamma')$ ($n \geq 1$) such that γ_i and γ'_i are not congruent in s for some $1 \leq i \leq n$.

   For every variable $x \in \mathcal{V}$, x is *bound* in $s \in \mathcal{S}$ iff for some $f \in \mathcal{F}$, $f(\gamma_1, \ldots, \gamma_n) \in s(x)$ ($n \geq 0$), where γ_i is a term for every $1 \leq i \leq n$.

2. $\Theta \subseteq \mathcal{S}$ is a set of initial congruence relations.

3. $\mathcal{B}$ is a set of multisets of atoms. For every $b \in \mathcal{B}$, if $a \in b$, then $a \in \mathcal{A}$.

4. $\Lambda \subseteq \mathcal{B}$ is a set of initial multisets of atoms.

5. Let $\mathcal{G}$ be the set of *guards* such that for every $g \in \mathcal{G}$, g is a monotone function $g : \mathcal{S} \rightarrow \{true, false, \perp\}$, where $\perp \preceq true$, $\perp \preceq false$. $g \in \mathcal{G}$ is monotone iff for every $s, s' \in \mathcal{S}$, $s \preceq s' \Rightarrow g(s) \preceq g(s')$.

   Let $true \in \mathcal{G}$ such that for every $s \in \mathcal{S}$, $true(s) = true$.

6. $\rightharpoonup \subseteq \mathcal{S} \times \mathcal{B} \times \mathcal{G} \times \mathcal{S} \times \mathcal{B}$ is a transition relation. A transition $(s, b, g, s', b') \in \rightharpoonup$ is also denoted $(s,b) \overset{g}{\rightharpoonup} (s', b')$. Every transition $(s,b) \overset{g}{\rightharpoonup} (s', b')$ satisfies the following two conditions: $s \preceq s'$, and for some $a \in \mathcal{A}$, $a \in b \wedge b' = b - \{a\} + \delta$, where δ is a multiset of atoms; the relation $s' - s$ is called a *constraint*, and δ is called a *definition* of a. For every $s \in \mathcal{S}$ and every $b \in \mathcal{B}$, there is an idle transition $\tau_I = (s,b) \overset{true}{\rightharpoonup} (s,b)$.

   A transition $(s,b) \overset{g}{\rightharpoonup} (s', b')$ is said to be *enabled* on a state s iff $g(s) = true$. The idle transition is enabled on every s.

Process For a transition system $\Psi = < \mathcal{S}, \Theta, \mathcal{B}, \Lambda, \rightharpoonup >$, a *process* is an atom $a \in \mathcal{A}$ that is defined by the definition of a in a transition $(s,b) \overset{g}{\rightharpoonup} (s', b')$.

Computation For a transition system $\Psi =< \mathcal{S}, \Theta, \mathcal{B}, \Lambda, \rightharpoonup >$, an infinite sequence $(\vec{s}, \vec{b}) = (s_0, b_0), (s_1, b_1), \ldots$, where for every $i \geq 0$, $s_i \in \mathcal{S}$ and $b_i \in \mathcal{B}$, is said to be a *computation* of Ψ iff the following conditions hold:

1. $s_0 \in \Theta$.

2. $b_0 \in \Lambda$.

3. For every $i \geq 0$, $(s_i, b_i) \overset{g}{\rightharpoonup} (s_{i+1}, b_{i+1})$ for some $g \in \mathcal{G}$ such that $g(s_i) = true$, in which case $(s_i, b_i) \overset{g}{\rightharpoonup} (s_{i+1}, b_{i+1})$ is said to be *taken* at position i.

For the computation $(\vec{s}, \vec{b})$, if for some i and every $j \geq i$, $(s_j, b_j) \overset{true}{\rightharpoonup} (s_{j+1}, b_{j+1}) = \tau_I$, then

1. (s,b) is said to *succeed* iff $b_i = \emptyset$, else

2. (s,b) is said to *fail* iff for every $(s_i, b_i) \overset{g}{\rightharpoonup} (s'_i, b'_i) \neq \tau_I$, $g(s_i) = false$, else

3. (s,b) is said to *deadlock*.

The notion of time is integrated with the transition systems in a similar way as in Timed Transition System [10]; a global fictitious clock is assumed, and transitions are assumed to be taken "instantaneously," while timing constraints restrict when transactions may take place.

Timed state sequence Let $\mathcal{R}$ be the set of nonnegative reals. Time is expressed as an infinite monotonic sequence over $\mathcal{R}$. A timed state sequence is defined as $\rho = (\vec{s}, \vec{b}, T)$, where $(\vec{s}, \vec{b}) = (s_0, b_0), (s_1, b_1), \ldots, (s_i, b_i), \ldots$ ($i \geq 0$) is an infinite sequence of state-processes pairs, and T is a sequence of the corresponding time values $T_i \in \mathcal{R}$, such that either $T_{i+1} = T_i$ or $T_{i+1} > T_i \wedge (s_{i+1}, b_{i+1}) = (s_i, b_i)$ (in other words, progress of time is interleaved with the change of states), and for every $t \in \mathcal{R}$, there is some i such that $t \leq T_i$.

Timed transition system for Flat TGHC A timed transition system Ψ^T is a tuple $< \mathcal{S}, \Theta, \mathcal{B}, \Lambda, \rightharpoonup, l, u >$, with the underlying transition system $\Psi =< \mathcal{S}, \Theta, \mathcal{B}, \Lambda, \rightharpoonup>$, lower bounds for transitions $l :\rightharpoonup \rightarrow \mathcal{N}$, and upper bounds for transitions $u :\rightharpoonup \rightarrow \mathcal{N} \cup \{\infty\}$, where $\mathcal{N}$ is the set of non-negative integers; for every $\tau = (s,b) \overset{g}{\rightharpoonup} (s',b')$, $l(\tau) \leq u(\tau)$, and $l(\tau_I) = 0$ and $u(\tau_I) = \infty$.

Timed computation For a timed transition system $\Psi^T =< \mathcal{S}, \Theta, \mathcal{B}, \Lambda, \rightharpoonup, l, u >$, a timed state sequence $\rho = (\vec{s}, \vec{b}, T)$ is said to be a *computation* of Ψ^T iff the sequence $(\vec{s}, \vec{b})$ is a computation of the underlying transition system Ψ, and the following conditions hold:

1. For every $\tau = (s_i, b_i) \overset{g}{\rightharpoonup} (s_{i+1}, b_{i+1})$ $(i \geq 0)$ and every $j \leq i$ such that $T_{i+1} - l(\tau) \leq T_j$, $g(s_j) = true$.
2. For every $\tau = (s_i, b_i) \overset{g}{\rightharpoonup} (s'_i, b'_i)$ $(i \geq 0)$, if τ is taken or disabled at position i, then for every $j \leq i$ such that $T_j + u(\tau) \leq T_i$, $g(s_j) = \perp$.

Intuitively, once a transition is enabled, it must be taken between the lower bounds and the upper bounds of the time measured from when the guards become true, or it must be disabled. Time intervals are denoted $[l, u]$ henceforth (*e.g.* for a transition τ with a time interval $l(\tau) = 0$ and $u(\tau) = \infty$, the time interval is denoted $[0, \infty]$).

A.2 Flat GHC

Flat GHC is a subset of *GHC*, which can be implemented efficiently; it is known that *Flat GHC* is sufficient for most applications. A simple operational semantics of *Flat GHC* programs is formulated by using the transition system for *Flat GHC*. For other formulations, readers are refered to [16, 29].

Flatness A guarded Horn clause is said to be *flat* iff whose guard goals contain only the following:

1. Unification goals.
2. Goals such that bodies of whose candidate clauses are all empty.

A *Flat GHC* program contains flat guarded Horn clauses only.

Flat Guarded Horn clause A *Flat GHC* program is a set of flat guarded Horn clauses, defined as follows:

$$H \leftarrow G_N, G_T \mid B_N, B_U$$

where H is an atom, and G_N,G_T,B_N, and B_U are multisets of atoms. H is the *head*. If H contains a predicate symbol p, the clause is said to *define* p. A predefined binary predicate to show an congruence relation over terms "=" is assumed. A goal with "=" is called a unification goal. G_N is a set of unification guard goals, and G_T is a set of unification guard goals that will be treated as the timed guard in the next subsection. B_N is a multiset of non-unification body goals, and B_U is a set of unification body goals.

In the above definition, H is assumed to be an atom of the following form:

$$p(x_1, \ldots, x_n) \ (n \geq 0)$$

where x_i denotes a variable for every $1 \leq i \leq n$; this does not restrict the use of the language since

$$p(\gamma_1, \ldots, \gamma_n)$$

where γ_i denotes a term for every $1 \leq i \leq n$, can be rewritten as

$$p(x_1, \ldots, x_n) \leftarrow x_1 = \gamma_1, \ldots, x_n = \gamma_n$$

without changing its meaning. Also, the following clauses

$$p(\ldots) \leftarrow q(x_1, \ldots, x_n), \ldots \mid \ldots$$

$$q(y_1, \ldots, y_n) \leftarrow y_1 = \gamma_1, \ldots, y_n = \gamma_n \mid true. \ (n \geq 0)$$

where x_i,y_i denote a variable and γ_i denotes a term for every $1 \leq i \leq n$, can be rewritten as

$$p(\ldots) \leftarrow x_1 = \gamma_1, \ldots, x_n = \gamma_n, \ldots \mid \ldots$$

so that non-unification goals in the guard can be replaced by unification goals; this simplifies the formulation of the semantics.

A program is invoked by a goal clause, defined as follows:

$$\leftarrow B_N, B_U$$

where B_N is a multiset of non-unification goals, and B_U is a multiset of unification goals.

Flat GHC program A *Flat GHC* program $\mathcal{C}$ (*i.e.* a set of flat guarded Horn clauses) is associated with a transition system $\Psi_{GHC} =< \mathcal{S}, \Theta, \mathcal{B}, \Lambda, \rightharpoonup>$, such that:

1. For every goal clause $\leftarrow B_N, B_U$ for $\mathcal{C}$, there is one $s \in \Theta$ such that $(\gamma, \gamma') \in s$ for every $\gamma = \gamma' \in B_U$.
2. For every goal clause $\leftarrow B_N, B_U$ for $\mathcal{C}$, $B_N \in \Lambda$.

Transition For a *Flat GHC* program $\mathcal{C}$ and the associated transition system $\Psi_{GHC} =< \mathcal{S}, \Theta, \mathcal{B}, \Lambda, \rightharpoonup>$, A guarded Horn clause $c = H \leftarrow G_N, G_T \mid B_N, B_U \in \mathcal{C}$, defining an n-ary predicate p, yields a transition $(s,b) \overset{g}{\rightharpoonup} (s',b')$ for each appropriate a such that:

1. process a defined in the transition is an atom of the form $p(\gamma_1, \ldots, \gamma_n)$.
2. There is a substitution clause $H^a \leftarrow G_N^a, G_T^a \mid B_N^a, B_U^a$ for c, obtained by the following substitution of variables in c:

   (a) Variables appearing in H – replace them with the corresponding terms in a.
   (b) Variables appearing in every $\gamma = \gamma' \in G_N \cup G_T$ – replace them with the corresponding terms in some $s(\gamma')$ if the variable appears in γ, and replace them with the corresponding terms in some $s(\gamma)$ if the variable appears in γ'; if corresponding terms do not exist, there is no substitution clause.
   (c) Other variables are replaced by variables unbound in s that never appeared in the arguments of atoms in the computation.

3. For every $\gamma = \gamma' \in G_N^a$, $(\gamma, \gamma') \in s$.

4. g is a function such that:
   (a) *true* iff for every $\gamma = \gamma' \in G_T^a$, $(\gamma, \gamma') \in s$, else
   (b) *false* iff for some $\gamma = \gamma' \in G_T^a$, γ and γ' are not congruent in s, else
   (c) $\perp$.

5. For every $\gamma = \gamma' \in B_U^a$, $(\gamma, \gamma') \in s'$ (if some γ and γ' are not congruent in s', then the computation containing the transition fails).

6. B_N^a is the definition of a.

Commitment Commitment is to select a transition to be taken; if there are more than one transition that can be taken, one of them is committed nondeterministically.

A.3 Flat TGHC

Flat timed guarded Horn clause A *Flat TGHC* program is a set of flat timed guarded Horn clauses, defined as follows:

$$H \leftarrow G_N :: G_T\ [\alpha, \beta] \mid B_N, B_U$$

where $H \leftarrow G_N, G_T \mid B_N, B_U$ is the corresponding flat guarded Horn clause, and $[\alpha, \beta]$ is the time interval $(\alpha, \beta \in \mathcal{N})$.

Flat TGHC program A *Flat TGHC* program $\mathcal{C}^T$ (*i.e.* a set of flat timed guarded Horn clauses) is associated with a timed transition system $\Psi_{GHC}^T =< \mathcal{S}, \Theta, \mathcal{B}, \Lambda, \rightharpoonup, l, u >$, such that:

1. $\Psi_{GHC} =< \mathcal{S}, \Theta, \mathcal{B}, \Lambda, \rightharpoonup, >$ is the underlying transition system corresponding to $\mathcal{C}$, such that every flat guarded Horn clause in $\mathcal{C}$ is the correspondence to the clauses in $\mathcal{C}^T$, and

2. For every flat timed guarded Horn clause $H \leftarrow G_N :: G_T\ [\alpha, \beta] \mid B_N, B_U \in \mathcal{C}^T$ and every transition $\tau = (s, b) \xrightharpoonup{g} (s', b')$ represented by $H \leftarrow G_N, G_T \mid B_N, B_U$, the time interval is defined as follows:
   (a) $l(\tau) = \alpha$.
   (b) $u(\tau) = \beta$.

   If a transition cannot meet its time interval, the computation fails.

References

[1] Rajeev Alur and Thomas A. Henzinger, "Real-time Logics: Complexity and Expressiveness," *1990 IEEE Fifth Annual Symposium On Logic In Computer Science*, IEEE Computer Society Press, pp.390-401

[2] Rajeev Alur, Costas Courcoubetis, and David Dill, "Model-Checking for Real-Time Systems," *1990 IEEE Fifth Annual Symposium On Logic In Computer Science*, IEEE Computer Society Press, pp.414-425

[3] J.L. Bergerand, P. Caspi, D. Pilaud, N. Halbwachs, and E. Pilaud, "Outline of a Real Time Data Flow Language," *IEEE 1985 Real-Time Systems Symposium*, IEEE Computer Society Press, pp.33-42

[4] Takashi Chikayama, Hiroyuki Sato, and Toshihiko Miyazaki, "Overview of the Parallel Inference Machine Operating System (PIMOS)," *Fifth Generation Computer Systems 1988*, Springer-Verlag, pp.230-251

[5] Keith Clark and Steve Gregory, "A Relational Language for Parallel Programming," *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, 1981, pp.171-178

[6] Keith Clark and Steve Gregory, "PARLOG: Parallel Programming in Logic," *Concurrent Prolog - Collected Papers - Volume 1*, MIT Press, 1987, pp.84-139

[7] Ian T. Foster, "Logic Operating Systems: Design Issues," *Logic Programming (Proceedings of the Fourth International Conference)*, The MIT Press, pp.910-926

[8] A.A. Franstini and E.B. Lewis, "A Declarative Language for the Specification of Real Time Systems," *IEEE 1985 Real-Time Systems Symposium*, IEEE Computer Society Press, pp.43-51

[9] Narain Gehani and Krithi Ramamritham, "Real-Time Concurrent C: A Language for Programming Dynamic Real-Time Systems," *The Journal of Real-Time Systems*, vol.3, no.4, Kluwer Academic Publishers, 1991, pp.377-405

[10] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli, *Timed Transition Systems*, Technical Report, Cornell University, TR 92-1263, 1992.

[11] N. Halbwachs, D. Pilaud, and F. Ouabdesselam, "Specifying, Programming and Verifying Real-Time Systems Using a Synchronous Declarative Language," *Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science 407, Springer-Verlag, 1989, pp.213-231

[12] J. van Katwijk (Ed.), *Ada: Moving Towards 2000*, Lecture Notes in Computer Science 603, Springer-Verlag, 1992

[13] Shmuel Kliger, Eyal Yardeni, Kenneth Kahn, and Ehud Shapiro, "The Language FCP(:,?)," *Fifth Generation Computer Systems 1988*, Springer-Verlag, pp.763-773

[14] Noboru Koshizuka, Hiroaki Takada, Masaharu Saito, Yasushi Saito, and Ken Sakamura, "Implementation Issues of the TACL/TULS Language System on BTRON," *TRON Project 1989*, Springer Verlag, pp.113-130

[15] Shem-Tov Levi and Ashok K. Agrawala, *REAL-TIME SYSTEM DESIGN*, McGraw-Hill Publishing Company, 1990

[16] Michael. J. Maher, "Logic Semantics for a Class of Committed-Choice Programs," *Logic Programming (Proceedings of the Fourth International Conference)*, The MIT Press, pp.858-876

[17] J.S. Ostroff, "A Verifier for Real-Time Properties," *The Journal of Real-Time Systems*, vol.4, no.1, Kluwer Academic Publishers, 1992, pp.5-35

[18] John A. Plaice, "RLucid, a general real-time dataflow language," *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science 571, Springer-Verlag, 1992, pp.363-374

[19] Shmuel Safra and Ehud Shapiro, "Meta Interpreters for Real," *Concurrent Prolog - Collected Papers - Volume 2*, MIT Press, 1987, pp.166-179

[20] Ken Sakamura, "The Computerized Society," *TRON Project 1989*, Springer Verlag, pp.3-13

[21] Ken Sakamura, "Programmable Interface Design in HFDS," *TRON Project 1990*, Springer Verlag, pp.3-21

[22] Ken Sakamura, "TRON Application Projects: Gearing Up for HFDS," *Proceedings of the Eighth TRON Project Symposium (International)*, IEEE Computer Society Press, 1991, pp.2-14

[23] Vijay A. Saraswat, "A Somewhat Logical Formulation of CLP Synchronization Primitives," *Logic Programming (Proceedings of the Fifth International Conference and Symposium)*, The MIT Press, pp.1298-1314

[24] Ehud Shapiro, "Concurrent Prolog: A Progress Report," *IEEE Computer* 19(8), 1986, pp.44-58

[25] Ehud Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," *Concurrent Prolog - Collected Papers - Volume 1*, MIT Press, 1987, pp.27-83

[26] Ehud Shapiro, "Systems Programming in Concurrent Prolog," *Concurrent Prolog - Collected Papers - Volume 2*, MIT Press, 1987, pp.6-27

[27] Ehud Shapiro (Ed.), *Concurrent Prolog - Collected Papers - Volume 1*, MIT Press, 1987

[28] Kazunori Ueda, "Guarded Horn Clauses," *Concurrent Prolog - Collected Papers - Volume 1*, MIT Press, 1987, pp.140-156

[29] Kazunori Ueda and Koichi Furukawa, "Transformation Rules for GHC Programs," *Fifth Generation Computer Systems 1988*, Springer-Verlag, pp.582-591

[30] Kazunori Ueda and Takashi Chikayama, "Design of the Kernel Language for the Parallel Inference Machine," *ICOT Journal* No.31, 1991

[31] Mark D. Wood, *Fault-Tolerant Management of Distributed Applications Using the Reactive System Architecture*, Ph.D Thesis, Cornell University, 1991