Projection in Temporal Logic Programming*

Zhenhua Duan, Maciej Koutny and Chris Holt

Department of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU, U.K.

October 19, 1995

Abstract

We define a projection operator in the framework of the temporal logic programming. Its syntax and semantics are presented and illustrated with examples. We also discuss the implementation details of the projection construct.

Keywords: Temporal logic, programming, projection.

1 Introduction

Temporal Logic Programming [6, 7, 9] is a paradigm for the specification and verification of sequential and concurrent programs. Within a temporal logic programming language, such as Tempura [6], the next, always and chop are useful operators for sequential programs, while conjunction and parallel composition are basic operators for concurrent programming. An advantage of the conjunction construct is its simplicity. However, it seems appropriate for dealing with fine-grained parallel operations that proceed in lock-step. The parallel composition operator ($\|$, see Section 2), on the other hand, permits the combined processes to specify their own intervals. Thus it is better suited to the coarse-grained concurrency of a typical multiprocessor, where each process proceeds at its own speed. Moreover, processes combined through the parallel composition operator share all the states and may interfere with one another. Therefore, it is interesting and desirable to investigate other ways of handling parallel computations which would combine some features of both conjunction and parallel composition operators.

Projection, $p\ proj\ q$, was originally employed for the purpose of modelling hardware assuming different granularities of time [6] (see Section 3). It requires that process p be repeatedly executed over a sequence of successive subintervals. This can be

*This research was partially supported by the SERC Grant 491105.

inconvenient because it is not always desirable to execute the same process several times. Moreover, it requires both processes p and q to terminate at the same time. In general, it is not the case that processes are executed so regularly.

In this paper we introduce a new projection operator, $(p_1, \ldots, p_m)\ prj\ q$, which can be thought of as a combination of the parallel and projection operators. Intuitively, it means that q is executed in parallel with $p_1; \ldots; p_m$ over an interval obtained by taking the endpoints (randezvous points) of the intervals over which $p_1, \ldots, p_m$ are executed. The projection construct permits the processes $p_1, \ldots, p_m, q$ to be autonomous, each process having the right to specify the interval over which it is executed. In particular, the sequence of processes $p_1, \ldots, p_m$ and process q may terminate at different time points. Although the communication between processes is still based on shared variables, the communication and synchronization only take place at the rendezvous points (global states), otherwise they are executed independently. The projection operator also enables the specification of program execution using different time scales.

The paper is organised as follows: The next section introduces the syntax and semantics of the temporal logic we use. In Section 3, the new projection operator is defined and some of its basic properties are shown. In Section 4, the implementation details of the new operator are discussed. Section 5 contains examples.

2 Logic Framework

Our underlying logic is the first order temporal logic [3, 5] with chop [1, 8], and is an extension of ITL [6].

2.1 Syntax

Let Π be a countable set of *propositions*, and V be a countable set of typed static and dynamic *variables*. The *terms* of the logic e and formulas f are given by the following grammar:[1]

$$\begin{array}{lcl} e & ::= & x \mid \bigcirc e \mid \ominus e \mid f_0 \mid f_1(e) \mid f_2(e,e) \mid \ldots \\ p & ::= & \pi \mid e = e \mid P_1(e) \mid P_2(e,e) \mid \ldots \mid \neg p \mid p \wedge p \mid \exists x : p \mid \bigcirc p \mid \ominus p \mid p;p \end{array}$$

In $f_i(e_1, \ldots, e_i)$ and $P_i(e_1, \ldots, e_i)$ it is assumed that the types of the terms are compatible with those of the arguments of f_i and P_i.

[1] x is a variable, f_i is a function of arity i, π is a proposition and P_i is an atomic predicate (different from equality) of arity i. In particular, f_0 is a constant term.

The derived connectives, $\vee$, $\rightarrow$ and $\leftrightarrow$, as well as the logic constants, $true$ and $false$, are defined as usual. We also use the following derived formulas:

$$
\begin{array}{lllllll}
empty & = & \neg \bigcirc true & & more & = & \neg empty \\[1em]
\Diamond p & = & true; p & & \Box p & = & \neg \Diamond \neg p \\[1em]
\odot p & = & \neg \bigcirc \neg p & & skip & = & len(1) \\[1em]
p \parallel q & = & p \wedge (q; true) \vee q \wedge (p; true) & & len(n) & = & \begin{cases} empty & n = 0 \\ \bigcirc len(n-1) & n > 1 \end{cases}
\end{array}
$$

The temporal operators are called *previous* ($\ominus$), *next* ($\bigcirc$), *chop* (;), *always* ($\Box$), *sometimes* ($\Diamond$), *weak next* ($\odot$) and *parallel* ($\parallel$).

2.2 Semantics

A *state* s is an assignment which for each variable $v \in V$ defines $s[v]$, and for each proposition $\pi \in \Pi$ defines $s[\pi]$. $s[v]$ is a value of the appropriate type or nil (undefined), whereas $s[\pi] \in \{true, false\}$.

An *interval* is a non-empty (possibly infinite) sequence of states, $\sigma = \langle s_0, s_1, \ldots \rangle$. The length of σ, denoted by $|\sigma|$, is defined as ω if σ is infinite; otherwise it is the number of states in σ minus 1. For $0 \leq i, j \leq |\sigma|$ we will use $\sigma_{(i..j)}$ to denote the subinterval $\langle s_i, s_{i+1}, \ldots, s_j \rangle$.[2] It is assumed that each static variable is assigned the same value in all the states in σ. The concatenation of a finite σ with another interval (or empty string) σ' is denoted by $\sigma \cdot \sigma'$.

An *interpretation* is a tuple $\mathcal{I} = (\sigma, i, k, j)$, where $\sigma = \langle s_0, s_1, \ldots \rangle$ is an interval, i and k are integers, and j is an integer or ω, such that $i \leq k \leq j \leq |\sigma|$. We use (σ, i, k, j) to mean that a formula or term is interpreted over a subinterval $\sigma_{(i..j)}$ with the current state being s_k.

For every term e, the evaluation of e relative to interpretation $\mathcal{I}$, $\mathcal{I}[e]$, is defined in the following way:

$$
\begin{array}{rcl}
\mathcal{I}[v] & = & s_k[v] \\[1em]
\mathcal{I}[f(e_1, \ldots, e_m)] & = & \begin{cases} f(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) & \text{if } \mathcal{I}[e_h] \neq nil \text{ for all } h \\ nil & \text{otherwise} \end{cases}
\end{array}
$$

[2]When $i > j$, $\sigma_{(i..j)}$ is the empty string.

$$\mathcal{I}[\bigcirc e] = \begin{cases} (\sigma, i, k+1, j)[e] & \text{if } k < j \\ nil & \text{otherwise} \end{cases}$$

$$\mathcal{I}[\odot e] = \begin{cases} (\sigma, i, k-1, j)[e] & \text{if } i < k \\ nil & \text{otherwise} \end{cases}$$

The satisfaction relation for formulas, $\models$, is defined as the least relation satisfying the following:

$\mathcal{I} \models true$

$\mathcal{I} \not\models false$

$\mathcal{I} \models \pi$ if $s_k[\pi] = true$.

$\mathcal{I} \models P(e_1, \ldots, e_m)$ if, for all h, $\mathcal{I}[e_h] \neq nil$ and $P(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) = true$.

$\mathcal{I} \models e = e'$ if $\mathcal{I}[e] = \mathcal{I}[e']$.

$\mathcal{I} \models \neg p$ if $\mathcal{I} \not\models p$.

$\mathcal{I} \models p \wedge q$ if $\mathcal{I} \models p$ and $\mathcal{I} \models q$.

$\mathcal{I} \models \bigcirc p$ if $k < j$ and $(\sigma, i, k+1, j) \models p$.

$\mathcal{I} \models \odot p$ if $i < k$ and $(\sigma, i, k-1, j) \models p$.

$\mathcal{I} \models p; q$ if there is an integer h, $k \leq h \leq j$, such that $(\sigma, i, k, h) \models p$ and $(\sigma, h, h, j) \models q$.

$\mathcal{I} \models \exists x : p$ if for some interval σ' which has the same length as σ, $(\sigma', i, k, j) \models p$ and the only difference between σ and σ' can be in the values assigned to variable x.

One can show that $\mathcal{I} \models p$ if and only if $(\sigma_{(i..j)}, 0, k-i, j-i) \models p$.[3] Moreover, if p is a formula which does not use the previous operator then $\mathcal{I} \models p$ if and only if $(\sigma_{(k..j)}, 0, 0, j-k) \models p$. If there is an interpretation $\mathcal{I}$ such that $\mathcal{I} \models p$ then p is *satisfiable*. If $\mathcal{I} \models p$, for all interpretations $\mathcal{I}$, then p is *valid*, denoted by $\Vdash p$.

We also define the satisfaction relation for intervals. Given an interval σ, $\sigma \models p$ if $(\sigma, 0, 0, |\sigma|) \models p$. Moreover, $\models p$ if $\sigma \models p$, for all intervals σ.

[3]That is the relevant part of σ in $\mathcal{I} = (\sigma, i, k, j)$ is $\sigma_{(i..j)}$. In particular, the valuations of variables and predicates outside the bounds given by i and j do not matter.

3 New Projection Operator

3.1 Syntax and Semantics

The new projection construct is defined as

$$(p_1, \ldots, p_m)\ prj\ q$$

where $p_1, \ldots, p_m$ and q are formulas ($m \geq 1$). To ensure smooth synchronization between $p_1, \ldots, p_m$ and q, in the implementation of the projection described in the next section, the previous operator is not allowed within q. However, it can be used in the p_l's. To define the semantics of the projection operator we need an auxiliary operator for intervals.

Let $\sigma = \langle s_0, s_1, \ldots \rangle$ be an interval and $r_1, \ldots, r_h$ be integers ($h \geq 1$) such that $0 \leq r_1 \leq r_2 \leq \ldots \leq r_k \leq |\sigma|$. The *projection* of σ onto $r_1, \ldots, r_h$ is the interval

$$\sigma{\downarrow}(r_1, \ldots, r_k) = \langle s_{t_1}, s_{t_2}, \ldots, s_{t_l} \rangle$$

where $t_1, \ldots, t_l$ is obtained from $r_1, \ldots, r_h$ by deleting all duplicates.[4] For example,

$$\langle s_0, s_1, s_2, s_3, s_4 \rangle{\downarrow}(0, 0, 2, 2, 2, 3) = \langle s_0, s_2, s_3 \rangle.$$

The semantics of the projection operator is defined, as before, relative to an interpretation $\mathcal{I} = (\sigma, i, k, j)$. Formally,

$$\mathcal{I} \models (p_1, \ldots, p_m)\ prj\ q$$

if $\sigma{\downarrow}(k) \models q$ and $\mathcal{I} \models p_1; \ldots; p_m$, or if there are integers $r_1, r_2, \ldots, r_h$ ($1 \leq h \leq m$) such that $k \leq r_1 \leq r_2 \leq \ldots \leq r_h \leq j$ and the following hold:

- $(\sigma, i, k, r_1) \models p_1$.
- For $1 < l \leq h$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l$.
- If $h < m$ then $\sigma{\downarrow}(k, r_1, \ldots, r_h) \models q$ and $(\sigma, r_h, r_h, j) \models p_{h+1}; \ldots; p_m$.
- If $h = m$ then $\sigma{\downarrow}(k, r_1, \ldots, r_h) \cdot \sigma_{(r_h+1..j)} \models q$.

In programming language terms, the interpretation of $(p_1, \ldots, p_m)\ prj\ q$ is somewhat sophisticated as we need *two* sequences of clocks (states) running on different time scales: one is a local state sequence, over which $p_1, \ldots, p_m$ are executed, the other is a global state sequence over which q is executed. Process q is executed in a parallel manner with the sequence of processes $p_1, \ldots, p_m$. The execution proceeds as follows (see Figure 1): First, q and p_1 start at the first global state and p_1 is executed over a

[4] $t_1, \ldots, t_l$ is the maximal strictly increasing subsequence of $r_1, \ldots, r_h$.

sequence of local states until its termination. Then (the remaining part of) q and p_2 are executed at the second global state. Subsequently, p_2 is continuously executed over a sequence of local states until its termination, and so on. Although q and p_1 start at the same time, $p_1, \ldots, p_m$ and q may terminate at different time points. If q terminates before some p_{h+1}, then, subsequently, $p_{h+1}, \ldots, p_m$ are executed sequentially. If $p_1, \ldots, p_m$ are finished before q, then the execution of q is continued until its termination.

```
t0        t2        t4        t6
|---------|---------|---------|
|<------------q-------------->|
t0   t1   t2   t3   t4   t5   t6   t7   t8
|----|----|----|----|----|----|----|----|
|<---p1-->|<---p2-->|<---p3-->|<---p4-->|

(a): q terminates before p4

t0        t2        t4        t6        t8   t9   t10
|---------|---------|---------|---------|----|----|
|<------------------q---------------------------->|
t0   t1   t2   t3   t4   t5   t6   t7   t8
|----|----|----|----|----|----|----|----|
|<---p1-->|<---p2-->|<---p3-->|<---p4-->|

(b): p4 terminates before q

t0        t2        t4        t6        t8
|---------|---------|---------|---------|
|<------------------q------------------>|
t0   t1   t2   t3   t4   t5   t6   t7   t8
|----|----|----|----|----|----|----|----|
|<---p1-->|<---p2-->|<---p3-->|<---p4-->|

(c): q and p4 terminate at the same point
```

Figure 1: Possible executions of (p1,p2,p3,p4) prj q

Projection can be thought of as a special parallel computation which is executed on different time scales. Consider the following formulas:

$$p_1 \stackrel{def}{=} len(2) \wedge \Box(more \rightarrow (\bigcirc i = i + 2))$$

$$p_2 \stackrel{def}{=} len(4) \wedge \Box(more \rightarrow (\bigcirc i = i + 3))$$

$$p_3 \stackrel{def}{=} len(6) \wedge \Box(more \rightarrow (\bigcirc i = i + 4))$$

$$q \stackrel{def}{=} len(4) \wedge (i = 2) \wedge (j = 0) \wedge \Box(more \rightarrow (\bigcirc j = j + i)).$$

Then executing $(p_1, p_2, p_3)\ prj\ q$ yields the following result:

```
  t0      t2              t6                      t12 t13
  |-------|---------------|-----------------------|---|
  |-----------------------q--------------------------->|
  t0  t1  t2  t3  t4  t5  t6  t7  t8  t9  t10 t11 t12
  |---|---|---|---|---|---|---|---|---|---|---|---|
  |<--p1->|<-----p2------>|<--------- p3--------->|
i=2   4   6   9   12  15  18  22  26  30  34  38  42
j=0       2               8                       26  68
```

Figure 2: Projection computation

The original projection operator defined in [6], $p\ proj\ q$, and the new projection operator defined above are not directly comparable. In the former, the formula p is executed repeatedly over a series of consecutive subintervals whose endpoints form the interval over which q is executed. This may result in repeating the same global state in the execution of q several times if some of the copies of p are executed over subintervals of zero length (in contrast, our definition in Section 3 rules this out). Moreover, in $p\ proj\ q$, the series of p's and the q always terminate at the same state. We feel that although $p\ proj\ q$ and $(p_1, \ldots, p_m)\ prj\ q$ do share some important properties, they still possess sufficiently distinct features to be treated independently as complementary constructs useful in the programming environment in which different time scales need to be considered.

3.2 Properties of Projection Operator

Projection enjoys a number of interesting properties. The theorem below is intended to formalize some of them. In what follows, a formula p is called *non-local* if for all σ, $\sigma \models p$ implies $|\sigma| \geq 1$.

Theorem 3.1

Let $p, q, p_1, \ldots, p_m$ be formulas.

1. $\models (empty\ prj\ q) \leftrightarrow q$.
2. $\models (q\ prj\ empty) \leftrightarrow q$.
3. $\models ((empty, p_1, \ldots, p_m)\ prj\ empty) \leftrightarrow (p_1; \ldots; p_m)$.
4. $\models (skip\ prj\ q) \leftrightarrow q$, *if q is non-local.*
5. $\models (q\ prj\ skip) \leftrightarrow q$, *if q is non-local.*
6. $\models ((p, q)\ prj\ skip) \leftrightarrow (p; q)$, *if p or q is non-local.*
7. $\models\vdash p \parallel q$
   $\leftrightarrow p \wedge ((empty, q, true)\ prj\ empty) \vee q \wedge ((empty, p, true)\ prj\ empty)$.
8. $\Vdash (p_1, \ldots, (p_i \vee p_i'), \ldots, p_m)\ prj\ q$
   $\leftrightarrow (p_1, \ldots, p_i, \ldots, p_m)\ prj\ q \vee (p_1, \ldots, p_i', \ldots, p_m)\ prj\ q$.
9. $\Vdash (p_1, \ldots, p_m)\ prj\ (p \vee q)$
   $\leftrightarrow (p_1, \ldots p_m)\ prj\ p \vee (p_1, \ldots, p_m)\ prj\ q$.

Proof: **(1)** Suppose $\sigma \models empty\ prj\ q$. If $\sigma{\downarrow}(0) \models q$ and $\sigma \models empty$ then $|\sigma| = 0$ and hence $\sigma \models q$. Otherwise, there is r, $0 \le r \le |\sigma|$, such that $(\sigma, 0, 0, r) \models empty$ and $\sigma{\downarrow}(0, r){\cdot}\sigma_{(r+1..|\sigma|)} \models q$. The former yields $r = 0$. Hence $\sigma \models q$.

Conversely, if $\sigma \models q$ then, by taking $r_1 = 0$, one can show that $\sigma \models empty\ prj\ q$.

(2) Suppose $\sigma \models q\ prj\ empty$. If $\sigma{\downarrow}(0) \models empty$ and $\sigma \models q$ then we are done. Otherwise, there is r, $0 \le r \le |\sigma|$, such that $(\sigma, 0, 0, r) \models q$ and $\sigma{\downarrow}(0, r){\cdot}\sigma_{(r+1..|\sigma|)} \models empty$. The latter means that $r = |\sigma| = 0$. Hence $\sigma \models q$.

Conversely, if $\sigma \models q$ then, since $\sigma{\downarrow}(0) \models empty$, $\sigma \models q\ prj\ empty$.

(3) Suppose $\sigma \models (empty, p_1, \ldots, p_m)\ prj\ empty$. If $\sigma{\downarrow}(0) \models empty$ and $\sigma \models empty; p_1; \ldots; p_m$ then clearly $\sigma \models p_1; \ldots; p_m$. Otherwise, there are integers $r_1, \ldots, r_h$ $(1 \le h \le m+1)$ such that $0 \le r_1 \le \ldots \le r_h \le |\sigma|$ and the following hold:

- $(\sigma, 0, 0, r_1) \models empty$
- For $1 < l \le h$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p_{l-1}$
- If $h < m+1$ then $\sigma{\downarrow}(0, r_1, \ldots, r_h) \models empty$ and $(\sigma, r_h, r_h, |\sigma|) \models p_h; \ldots; p_m$.
- If $h = m+1$ then $\sigma{\downarrow}(0, r_1, \ldots, r_h){\cdot}\sigma_{(r_h+1..|\sigma|)} \models empty$.

We first observe that $r_1 = 0$. Moreover, if $h < m+1$ then $r_1 = \cdots = r_h = 0$ and hence $\sigma \models p_1; \ldots; p_m$. If $h = m+1$ then $|\sigma| = r_1 = \cdots = r_h = 0$, yielding $\sigma \models p_1; \ldots; p_m$.

Conversely, if $\sigma \models p_1; \ldots; p_m$ then, by taking $h = 1$ and $r_1 = 0$, one can show that $\sigma \models (empty, p_1; \ldots; p_m)\ prj\ empty$.

(4) Suppose $\sigma \models skip\ prj\ q$. We first observe that $\sigma{\downarrow}(0) \models q$ and $\sigma \models skip$ is impossible since q is non-local. Hence there is r, $0 \le r \le |\sigma|$, such that $(\sigma, 0, 0, r) \models skip$ and $\sigma{\downarrow}(0, r){\cdot}\sigma_{(r+1..|\sigma|)} \models q$. The former implies $r = 1$ and hence $\sigma \models q$.

Suppose $\sigma \models q$. Then, since q is non-local, $|\sigma| \ge 1$. Hence $\sigma \models skip\ prj\ q$ can be shown by taking $r_1 = 1$.

(5) Suppose $\sigma \models q\ prj\ skip$. We first observe that $\sigma{\downarrow}(0) \models skip$ and $\sigma \models q$ is impossible. Hence there is r, $0 \le r \le |\sigma|$, such that $(\sigma, 0, 0, r) \models q$ and $\sigma{\downarrow}(0, r){\cdot}\sigma_{(r+1..|\sigma|)} \models skip$. The former implies $r \ge 1$ (since q is non-local). This and the latter means that $r = |\sigma|$. Hence $\sigma \models q$.

Conversely, if $\sigma \models q$ then, by taking $r_1 = |\sigma|$ we obtain $(\sigma, 0, 0, r_1) \models q$ and $\sigma{\downarrow}(0, r_1) \models skip$. Note that the latter follows from $r_1 \ge 1$ (q is non-local).

(6) Suppose $\sigma \models (p, q)\ prj\ skip$. We first observe that $\sigma{\downarrow}(0) \models skip$ and $\sigma \models p; q$ is impossible. Thus one of the following must hold:

- There is r, $0 \le r \le |\sigma|$, such that $(\sigma, 0, 0, r) \models p$, $\sigma{\downarrow}(0, r) \models skip$ and $(\sigma, r, r, |\sigma|) \models q$. Hence $\sigma \models p; q$.
- There are integers r_1, r_2 such that $0 \le r_1 \le r_2 \le |\sigma|$, $(\sigma, 0, 0, r_1) \models p$, $(\sigma, r_1, r_1, r_2) \models q$ and $\sigma{\downarrow}(0, r_1, r_2){\cdot}\sigma_{(r_2+1..|\sigma|)} \models skip$. Since at least one of p and q is non-local, we must have $r_1 \ge 1$ or $r_2 \ge r_1 + 1$. Thus, from $\sigma{\downarrow}(0, r_1, r_2){\cdot}\sigma_{(r_2+1..|\sigma|)} \models skip$ it follows that $|\sigma| = r_2$. Hence $\sigma \models p; q$.

Conversely, suppose $\sigma \models p; q$. Then there is l, $0 \le l \le |\sigma|$ such that $(\sigma, 0, 0, l) \models p$ and $(\sigma, l, l, |\sigma|) \models q$. Since at least one of p and q is non-local, either $l \ge 1$ or $|\sigma| > 0 = l$. If the former holds then, by taking $h = 1$ and $r_1 = l$, one can show that $\sigma \models (p, q)\ prj\ skip$. If the latter holds then one can come to the same conclusion by taking $h = 2$, $r_1 = 0$ and $r_2 = |\sigma|$.

Finally, (7) follows from (3), and (8,9) follow directly from the definition of the semantics of the projection operator and

$$\vdash q_1; \ldots; (q_j \vee q_j'); \ldots; q_s \leftrightarrow (q_1; \ldots; q_j; \ldots; q_s) \vee (q_1; \ldots; q_j'; \ldots; q_s).$$

■

4 Implementation of Projection Operator

The programming language we used is a subset of the underlying logic. It is an extension of the Tempura [6] augmented with framing, parallel, projection, and await operators [2]. In addition, the variables within a program are allowed to refer to their previous values. The negation of temporal formulas, being fundamentally non-deterministic, is not a primitive operator of the language. Instead, the conditional, *more* and *empty*, all defined in terms of negation, are taken as primitives. Programs can use several kinds of expressions, employing equality, conditional, assignment and iterative operators.

To implement the projection operator, we have developed a new interpreter using the SICSTUS Prolog.

4.1 Implementation strategy

The implementation is based on the tableau methods, i.e. to execute a formula is to transform it to a logically equivalent conjunction of two formulas, *Present* and *Remains*, where the former consists of immediate assignments to program variables, output of program variables, *true*, *false*, *more* and *empty*. The role of *more* and *empty* is to indicate whether or not the interval is terminated. The *Remains* is what is executed in the subsequent state (if any). It is in a *reduced form* if it only consists of conjuncts with a leading weak next operator. When preparing the execution of the next state, the procedure *next_w* is used to remove these weak next operators from the conjuncts and what is actually executed at the next state is the resulting formula, *Next*. Formally,

$$\begin{aligned} Present &= \textstyle\bigwedge_{i=1}^{m} present_i \\ Remains &= \textstyle\bigwedge_{i=1}^{n} \odot\, w_i \\ Next &= \textstyle\bigwedge_{i=1}^{n} w_i \end{aligned}$$

where each w_i is a Tempura formula and

$$present_i \quad ::= \quad x = a \mid display(a) \mid true \mid more \mid empty.$$

4.2 Done flag

The interpreter employs several flags to manage the reductions. One of the important ones, *done* flag, indicates whether or not an interval is terminated. At the beginning of the execution of each state, *done* flag is set to *nil*, and then the interpreter sets its value to either *true* or *false*, depending on whether the *more* or *empty* conjunct has been encountered. If the program fails to specify properly the interval for the program, the interpreter cannot set the *done* flag; it remains equal to *nil*. Thus an error is detected and indicated.

4.3 Program structure

The execution of a Tempura program proceeds through a number of states. The execution at a state is composed of reductions in several passes. After the last pass, the executed formula is reduced to the $Present \wedge Remains$ form. In fact, $Present$ is dissolved during the reduction. Its effect is reflected in updating and displaying the values of variables, setting the *done* flag, etc. What remains after the last pass of the reduction is $Remains$ which is executed at the next state if the interval over which the formula is executed is not yet finished.

4.4 Reduction of projection

The projection construct $(p_1, \ldots, p_m)\ prj\ q$ is implemented as follows: It is processed by first allocating a *done* flag initialized to *nil* to serve as a *done* flag for projected interval on which the statement q is executed and transforming the statement to the internal construct IC,

$$IC \ = \begin{cases} project((p_2, \ldots, p_m), p_1, q, done(nil)) & \text{if } m > 1 \\ \\ project(empty, p_1, q, done(nil)) & \text{if } m = 1 \end{cases}$$

which is immediately re-reduced. The construct $project(R, P, Q, done(D1))$ is executed by first saving the current doneflag to OLD and setting the doneflag to $done(D1)$. The statement Q is then reduced in the context to a new statement Q'. Afterwards, the current doneflag is saved to $D2$ and the old doneflag, $done(OLD)$, is restored, and the statement P is then reduced in the context to a new statement P'. If P' or Q' is not fully reduced, the overall *project* statement is rewritten as $project(R, P', Q', done(D2))$. This is returned as the result of the reduction. On the other hand, if P' and Q' are both fully reduced then, with the notation

$$\begin{array}{lcl} D & = & next\_w(P') \\ E & = & next\_w(Q') \\ choose(R1, (R2)) & = & (R1; choose((R2))) \\ choose(R1) & = & R1 \end{array}$$

the overall *project* statement is transformed as follows:

if $done(D2) = done(true)$
then
  if $R = (R1, (R2))$
  then $\bigcirc\,(D; choose(R1, (R2)))$
  else
    if $R = R1$
    then $\bigcirc\,(D; R1)$
    else
      if $R = empty$
      then $\bigcirc\, D$
else
  if $done(D2) = done(false)$
  then
    if $R = (R1, (R2))$
    then $\bigcirc\,(D; project((R2), R1, E, done(nil)))$
    else
      if $R = R1$
      then $\bigcirc\,(D; project(empty, R1, E, done(nil)))$
      else
        if $R = empty$
        then $\bigcirc\,(D; project(, empty, empty, E, done(nil)))$

This tests the done flag indexed by $done(D2)$. If it is true, the interval in which Q was reduced is finished and therefore the $next\_w(P')$ is executed followed by remaining formulas $(R1, (R2))$, chosen by the procedure *choose*, if they were not empty. On the other hand, if $done(D2) = done(false)$ the interval in which Q was executed is not yet finished. Therefore, the formula D is executed followed by the resumption of the projection statement.

5 Examples

We now present two simple applications of the projection construct. The first is a pulse generator for variable x which can assume two values: 0 (low) and 1 (high). We first define two types of processes: The first one is $hold(i)$ $(i \geq 1)$ which is executed over an interval of length i and ensures that the value of x remains constant in all but final state:

$$hold(i) \stackrel{def}{=} len(i) \wedge \Box(\bigcirc more \rightarrow (\bigcirc x = x)).$$

The other is $switch(j)$ which ensures that the value of x is first set to 0 and then changed at every subsequent state:

$$switch(j) \stackrel{def}{=} (x = 0) \wedge len(j) \wedge \Box(more \rightarrow (\bigcirc x = 1 - x)).$$

Having defined $hold(i)$ and $switch(j)$ we can define pulse generators with varying number and length of low and high intervals for x,

$$pulse(i_1, \ldots, i_k) \stackrel{def}{=} (hold(i_1), \ldots, hold(i_k)) \; prj \; switch(k).$$

The second example is that of special parallel computation. Consider the formula $((len(i_1), len(i_2), \ldots, len(i_k)) \; prj \; q) \wedge p$. This allows processes p and q to be executed in a special parallel manner in which p is executed over a series of subintervals, and q is executed at their endpoints:

```
t0      t2              t6              t10
|-------|---------------|---------------|
|-----------------------q-------------->|
t0  t1  t2  t3  t4  t5  t6  t7  t8  t9 t10 t11 t12
|---|---|---|---|---|---|---|---|---|---|---|---|
|<--------------------- p---------------------->|
```

Figure 3: Special parallel computation

6 Conclusion

The projection operator $(p_1, \ldots, p_m) \; prj \; q$ presented in this paper is rather powerful. For example, it can be used to specify computations on different time scales. We feel it has a potential of being a useful operator in temporal logic programming. Another possible application area is that of the real time systems. In this case, we could treat $p_1, \ldots, p_m$ as formulas over a series of dense or real intervals and q as a formula over a projected discrete interval.

References

[1] Barringer H., Kuiper R. and Pnueli A.: *Now you may compose temporal logic specifications.* Proceedings of 16th ACM Symposium on Theory of Computing, 51–63 (1984).

[2] Duan Z., Holt C. and Moszkowski B.: *An interpreter for an executable subset of extended interval temporal logic with framing and concurrent operators.* BCTCS 8, Newcastle upon Tyne, March (1992).

[3] Kröger F.: *Temporal logic of programs.* Springer-Verlag (1987).

[4] Lamport L.: *The temporal logic of actions.* Digital, System Design Center, December (1991).

[5] Manner Z. and Pnueli A.: *The temporal logic of reactive and concurrent systems.* Springer-Verlag (1992).

[6] Moszkowski B.: *Executing temporal logic programs.* Cambridge University Press Cambridge (1986).

[7] Ness L.: *L.0: A Parallel executable temporal logic language.* Proceeding of the ACM SIGSOFT, International Workshop on Formal Methods in Software Development, Napa, California, May (1990).

[8] Rosner R. and Pnueli A.: *A choppy logic.* First Annual IEEE Symposium on Logic In Computer Science, LNCS, Springer Verlag, 306–314 (1986).

[9] Tang Z.: *Toward a unified logic basis for programming languages.* Proceedings of IFIP Congress 83, Amsterdam, Elsevier Science Publishers B.V. (North-Holland), 425–429 (1983).