Appeared in the Proceedings of the Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software (ARTS'97), LNCS 1231, p. 79--94, Mallorca, Spain, May 21-23, 1997

Refining Interval Temporal Logic Specifications*

Antonio Cau and Hussein Zedan

Science and Engineering Research Centre,
Department of Computer Science, De Montfort University,
The Gateway, Leicester LE1 9BH, UK

Abstract. Interval Temporal Logic (ITL) was designed as a tool for the specification and verification of systems. The development of an executable subset of ITL, namely Tempura, was an important step in the use of temporal logic as it enables the developer to check, debug and simulate the design. However, a design methodology is missing that transforms an abstract ITL specification to an executable (concrete) Tempura program. The paper describes a development technique for ITL based on refinement calculus. The technique allows the development to proceed from high level "abstract" system specification to low level "concrete" implementation via a series of correctness preserving refinement steps. It also permits a mixture of abstract specification and concrete implementation at any development step.
To allow the development of such a technique, ITL is extended to include modularity, resources and explicit communication. This allows synchronous, asynchronous and shared variable concurrency to be explicitly expressed. These constructs also help in solving the problems, like lack of expressing modularity, timing and communication, discovered during the use of ITL and Tempura for a large-scale application[2].

1 Introduction

Interval Temporal Logic (ITL) was designed particularly as a formalism for the specification and design of systems [7,9]. The development of an executable subset of ITL, known as Tempura, [8], was a welcome step in the use of ITL as it enables the developer to check, debug and simulate the design.

The development technique employed within ITL is simply that the design/-implementation is done in Tempura. Properties are then expressed as formulae in ITL and proved using its associated proof system. This technique may be adequate for small sequential systems but for medium-to-large scale systems it becomes tedious, and inapplicable in the presence of concurrency. In order to deal with medium-to-large scale systems, Moszkowski [9] considered the issue of compositionality and introduced compositional proof rules based on the notion

* Funded by EPSRC Research Grant GR/K25922: A Compositional Approach to the Specification of Systems using ITL and Tempura.
E-mail {cau,zedan}@dmu.ac.uk

of *import* and *export* of formulae. But the problem of how to design such systems remains. Recent study, [2], also revealed shortcomings in the formalism in terms of modularity and explicit expressibility of timing constraints, resources and concurrency. In this paper, we deal with these shortcomings of ITL by providing a timed-communication model allowing explicit representation of concurrency, resources and timing. The proposed model is very closely related to that of TAM [11]. It requires the introduction of timed-communication, a timeout and resource allocation constructs. These TAM constructs are given an ITL semantics because the original semantics is not accessible enough.

In addition, we describe an alternative development technique for ITL based on the refinement calculus of [1,6]. In this technique, we distinguish between two level of representation. The first is "abstract" and the second is "concrete". At the first level, systems are specified at its highest level of abstraction where design and implementation issues are ignored. These issues are only considered during the transition from the abstract level to the concrete level. This transition is performed by correctness preserving refinement laws. During this transition, both abstract and concrete representations are allowed to intermix. Compositionality is achieved by ensuring the monotonicity of the constructs used. The representation at the abstract level is done using only pure ITL primitives (using the newly defined specification statement). At the concrete level, Tempura's constructs in addition to the newly defined ones are used.

A *timed-communication* command interacts solely at a particular moment with its environment. *Timeout* allows the control passing from one of its components to another when the former fails to engage in a given period of time. *Resource* allocation constructs request or release a certain amount of a particular resource. Following traditional approaches, the proposed model presents the concept of *abstract time* to relate communication events of parallel components of a system to a *global clock*. The abstract time device is basically used to control the execution of processes by means of various delay constraints such as *start delay* [10]. On the other hand, we want to stress the fact that an action (albeit visible or not) performed by a machine are not always instantaneous, but may need a certain amount of machine time to complete (small as this may be). This observation gives rise to the notion of *resource time* as a complementary to abstract time. The resource time mechanism is largely related to the issue of scheduler design in time-critical applications. In order to allow a proper treatment of *idle* and *busy* state of a process, it is important to distinguish between the two notions of time. In addition, our model is required to be compositional.

The computational model we adopt is essentially that of TAM. A model of computation defines mathematically an abstract architecture upon which applications will execute. A *system* is a collection of *agents* (which is our unit of computation), possibly executing concurrently and communicating (a)synchronously via communication links. Systems can themselves be viewed as single agents and composed into larger systems. Systems have timing constraints imposed at three levels; system wide communication deadlines, agent deadlines and sub-computation deadlines (within the computation of an individual agent). Dead-

lines may be dynamic i.e. dependent on results of the computation, or static. Imprecise computations may have a deadline dependent on the precision of the data they have to process. Deadlines are all considered to be hard, i.e., can not be missed. A system has a static configuration, and it must have at most a finite number of communication links and agents.

At any instant in time a system can be thought of as having a unique *state*. The system state is defined by the values in the communication links and the state variables of the system, the so called *frame*. This frame defines the variables that can possibly change during system executing, the variables outside this frame will certainly not change. *Computation* is defined to be a sequence of system states , i.e., an *interval* of states. An agent is described by a computation which may transform a local data-space and may read and write to communication links during execution. The local data-space for the agent is created when the agent starts execution and is destroyed when the agent terminates. No agent may read or write another agent's local data-space. The computation may have both minimum and maximum execution times imposed. An agent may perform both computation and communication. Only an agent which performs no computation or communication may terminate instantaneously, i.e., the so-called empty agent. An agent may start execution as a result of either a condition of the current time or a write event occurring on a specific communication link. These two conditions may be used to model periodic and sporadic tasks respectively.

An agent may write to at most a finite number of communication links and read from at most a finite number of them. Synchronous communication links, i.e., read and write occur at the same time, are called *channels*. Asynchronous communication links are called *shunts*. Synchronous communication links are modeled by a shared variable which contains three values: the first one indicates if there is an agent willing to read from the channel, the second one if there is an agent willing to write to the channel and the third is the value transmitted over the channel. Asynchronous communication links are modeled by a shared variable which contains two values: the first one is a stamp which is increased by one each time a new write to the shunt takes place, and the second one is the value which was most recently written. Shunt writing is destructive, shunt reading is not. Communication link readership may be restricted to a set of agents. These agents can then be considered as a subsystem where communication links which are read or written by the agents within the subsystem define the subsystem's boundary. Subsystems may not overlap. The stamps within the shunts enable the reading agents to compute according to the *freshness* of the data. The need for stamps in shunts is a direct consequence of the decision to use non-destructive asynchronous communication. When an agent performs two consecutive inputs from a shunt, if it reads the same data item twice it may need to know if each value is a result of two different writes or a single write.

The paper is organized as follows. Section 2 defines our semantic domain based on that of ITL. Section 3 introduces the specification statement. The new constructs are given in Section 4 together with their specification-oriented semantics. Section 5 is devoted to the study of algebraic properties of the new

constructs. Section 6 uses the new constructs in the well known watchdog timer example. Conclusion and future work are given in Section 7.

2 Semantic Model

An agent will be described by a set of computations. A computation is a sequence of states wherein the agent can be. In TAM this set of computations is described by a first order formula which tend to be very large and thus unreadable. ITL enables us to describe this set of computations in an eloquent way. We will first introduce the syntax and semantics of ITL and then introduce the extensions needed for the definition of the new concrete constructs.

2.1 Interval Temporal Logic

This section describes the syntax and formal semantics of ITL. This formal semantics is different from [9] in that it also deals with infinite intervals.

An interval σ is considered to be a (in)finite sequence of states $\sigma_0\sigma_1\ldots$, where a state σ_i is a mapping from the set of variables Var to the set of values Val. The length $|\sigma|$ of an interval $\sigma_0\ldots\sigma_n$ is equal to n (one less than the number of states in the interval, i.e., a one state interval has length 0).

The syntax of ITL is defined in Table 1 where i is a constant, a is a static variable (doesn't change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol, p is a predicate symbol.

Table 1. Syntax of ITL

Expressions
$exp ::= i \mid a \mid A \mid g(exp_1, \ldots, exp_n) \mid \imath a\colon f$
Formulae
$f ::= p(exp_1, \ldots, exp_n) \mid exp_1 = exp_2 \mid exp_1 < exp_2 \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathsf{skip} \mid f_1 ; f_2 \mid f^*$

The informal semantics of the most interesting constructs are as follows:

- $\imath a\colon f$: the value of a such that f holds.
- skip: unit interval (length 1).
- $f_1 ; f_2$: holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval.
- f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds.

The formal semantics is as follows: Let χ be a choice function which maps any nonempty set to some element in the set. We write $\sigma \sim_v \sigma'$ if the intervals σ and σ' are identical with the possible exception of their mappings for the variable v.

– $\mathcal{M}_\sigma[\![v]\!] = \sigma_0(v)$.
– $\mathcal{M}_\sigma[\![g(exp_1, \ldots, exp_n)]\!] = \hat{g}(\mathcal{M}_\sigma[\![exp_1]\!], \ldots, \mathcal{M}_\sigma[\![exp_n]\!])$.
– $\mathcal{M}_\sigma[\![\imath a\colon f]\!] = \begin{cases} \chi(u) & \text{if } u \neq \{\} \\ \chi(Val_a) & \text{otherwise} \end{cases}$
where $u = \{\sigma'(a) \mid \sigma \sim_a \sigma' \wedge \mathcal{M}_{\sigma'}[\![f]\!] = \mathrm{tt}\}$
– $\mathcal{M}_\sigma[\![p(exp_1, \ldots, exp_n)]\!] = \mathrm{tt}$ iff $\hat{p}(\mathcal{M}_\sigma[\![exp_1]\!], \ldots, \mathcal{M}_\sigma[\![exp_n]\!])$.
– $\mathcal{M}_\sigma[\![\neg f]\!] = \mathrm{tt}$ iff $\mathcal{M}_\sigma[\![f]\!] = \mathrm{ff}$.
– $\mathcal{M}_\sigma[\![f_1 \wedge f_2]\!] = \mathrm{tt}$ iff $\mathcal{M}_\sigma[\![f_1]\!] = \mathrm{tt}$ and $\mathcal{M}_\sigma[\![f_2]\!] = \mathrm{tt}$.
– $\mathcal{M}_\sigma[\![\forall v \bullet f]\!] = \mathrm{tt}$ iff for all σ' s.t. $\sigma \sim_v \sigma'$, $\mathcal{M}_{\sigma'}[\![f]\!] = \mathrm{tt}$.
– $\mathcal{M}_\sigma[\![\mathsf{skip}]\!] = \mathrm{tt}$ iff $|\sigma| = 1$.
– $\mathcal{M}_\sigma[\![f_1 \,;\, f_2]\!] = \mathrm{tt}$ iff
(exists a k, s.t. $\mathcal{M}_{\sigma_0 \ldots \sigma_k}[\![f_1]\!] = \mathrm{tt}$ and
((σ is infinite and $\mathcal{M}_{\sigma_k \ldots}[\![f_2]\!] = \mathrm{tt}$) or
(σ is finite and $k \leq |\sigma|$ and $\mathcal{M}_{\sigma_k \ldots \sigma_{|\sigma|}}[\![f_2]\!] = \mathrm{tt}$)))
or (σ is infinite and $\mathcal{M}_\sigma[\![f_1]\!]$).
– $\mathcal{M}_\sigma[\![f^*]\!] = \mathrm{tt}$ iff
if σ is infinite then
(exist $l_0, \ldots, l_n$ s.t. $l_0 = 0$ and $\mathcal{M}_{\sigma_{l_n} \ldots}[\![f]\!] = \mathrm{tt}$ and
for all $0 \leq i < n$, $l_i <= l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i} \ldots \sigma_{l_{i+1}}}[\![f]\!] = \mathrm{tt}$.)
or
(exist an infinite number of l_i s.t. $l_0 = 0$ and
for all $0 \leq i$, $l_i <= l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i} \ldots \sigma_{l_{i+1}}}[\![f]\!] = \mathrm{tt}$.)
else
(exist $l_0, \ldots, l_n$ s.t. $l_0 = 0$ and $l_n = |\sigma|$ and
for all $0 \leq i < n$, $l_i <= l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i} \ldots \sigma_{l_{i+1}}}[\![f]\!] = \mathrm{tt}$.)

Frequently used abbreviations are listed in table 2 and some concrete constructs are listed in Table 3.

Table 2. Frequently used abbreviations

$\bigcirc f$	$\widehat{=}\, \mathsf{skip} \,;\, f$	next f
inf	$\widehat{=}\, true \,;\, false$	infinite interval
$finite$	$\widehat{=}\, \neg inf$	finite interval
$\Diamond f$	$\widehat{=}\, finite \,;\, f$	(sometimes f)
$\Box f$	$\widehat{=}\, \neg\Diamond\neg f$	always f
$\text{ⓦ} f$	$\widehat{=}\, \neg \bigcirc \neg f$	weak next f
$\text{◈} f$	$\widehat{=}\, finite \,;\, f \,;\, true$	some subinterval
$\text{🄰} f$	$\widehat{=}\, \neg \text{◈} \neg f$	all subintervals
$more$	$\widehat{=}\, \bigcirc true$	non-empty interval
$fin\ f$	$\widehat{=}\, \Diamond(\mathsf{empty} \supset f)$	f is true in final state
$halt\ f$	$\widehat{=}\, \Box(\mathsf{empty} \equiv f)$	terminate interval when f
$keep\ f$	$\widehat{=}\, \text{🄰}(\mathsf{skip} \supset f)$	all unit subintervals
$fstar\ f$	$\widehat{=}\, (finite \wedge (finite \wedge f)^*) \,;\, (\mathsf{empty} \vee (inf \wedge f))$	finite chopstar
f^ω	$\widehat{=}\, inf \wedge (finite \wedge f)^*$	omega chopstar

Table 3. Some concrete constructs

$\mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2$	$\widehat{=} (f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$	if then else
empty	$\widehat{=} \neg more$	empty interval
$\mathsf{while}\ f_0\ \mathsf{do}\ f_1$	$\widehat{=}\ fstar\,(f_0 \wedge f_1) \wedge fin\ \neg f_0$	while loop
$\mathsf{for}\ n\ \mathsf{times\ do}\ P$	$\widehat{=} \mathsf{if}\ n = 0\ \mathsf{then\ empty\ else}\ P\ ;\ \mathsf{for}\ n-1\ \mathsf{times\ do}\ P$	for loop
$\bigcirc exp$	$\widehat{=} \imath a{:}\, \bigcirc(exp = a)$	next expression
$fin\ exp$	$\widehat{=} \imath a{:}\, fin\,(exp = a)$	final expression
$exp_1 := exp_2$	$\widehat{=} \bigcirc exp_1 = exp_2$	assignment
$exp_1 \leftarrow exp_2$	$\widehat{=} finite \wedge (fin\ exp_1) = exp_2$	temporal assignment
$\mathsf{stable}\ exp$	$\widehat{=} exp\ gets\ exp$	stability
$padded\ exp$	$\widehat{=} (\mathsf{stable}\,(exp)\ ;\ \mathsf{skip}) \vee \mathsf{empty}$	padded expression
$exp_1\ gets\ exp_2$	$\widehat{=} keep\,(exp_1 \leftarrow exp_2)$	gets
$exp_1 \lessapprox exp_2$	$\widehat{=} (exp_1 \leftarrow exp_2) \wedge padded\ exp_1$	padded temporal assignment
$intlen\,(exp)$	$\widehat{=} \exists I \bullet (I = 0) \wedge (I\ gets\ I + 1) \wedge fin\,(I = exp)$	interval length
len	$\widehat{=} \imath n{:}\, intlen\,(n)$	length

2.2 Extensions

Let W be a set of state variables then $frame(W)$ denotes that only the variables in W can possible change, i.e., the variables outside the frame don't change. The semantics is defined as followed:

- $\mathcal{M}_\sigma[\![\mathsf{frame}\,(W)]\!] = \mathrm{tt}$ iff for all $v \in Var - W$, $\mathcal{M}_\sigma[\![\mathsf{stable}\,(v)]\!]$.

Here, we adopt a combined state-communication model for the system behavior where the observables correspond to the following variables:

- The normal state variables of ITL.
- variables $C \in \mathrm{Chan}$, representing channels whose values are triples (wtr, wtw, v) where
  - wtr is a boolean value indicating whether the system is ready to accept a message from that channel.
  - wtw is a boolean value indicating whether the system is willing to send a message to that channel.
  - v is standing for the value currently in channel C when wtr and wtw are both true.
- variables s representing shunts whose values are tuples (t, v) where t is a stamp and v the value written.
- variable res standing for the resource res.

The domain of the variable $time$ is a linear order $(TIME, <, +, 0)$, where 0 is the least element, and $+$ is an addition operator.

3 Specification Statement

The syntax of specification statement is $W : f$ where W is a set of variables and f an ITL formula.

The specification statement represents a blackbox description of the behavior of the required system. When we specify agents that require a minimum execution interval, care must be taken as regard to the feasibility of the specification. This is to ensure that the written specification indeed conforms with whatever restricted computational (executable) model chosen.

The semantics of the specification statement is simply given as

$$W : f \mathrel{\widehat{=}} \mathsf{frame}\,(W) \wedge f$$

4 New concrete constructs

This section introduces new concrete constructs for reasoning about communication, timing and resource allocation.

4.1 Channel communication

Let C be a channel then $\mathsf{channel}\ C \in P$ denotes that a new channel is introduced. $C!e$ denotes an output agent that sends the value of expression e over C. $C?x$ denotes an input agent that stores the value received over C in x.

$$\begin{array}{ll}
\mathsf{channel}\ C\ \mathsf{in}\ P & \mathrel{\widehat{=}} \exists C \bullet P \\
C? & \mathrel{\widehat{=}} \Pi_1(C) = \mathit{true} \\
C! & \mathrel{\widehat{=}} \Pi_2(C) = \mathit{true} \\
C.x & \mathrel{\widehat{=}} \Pi_3(C) = x \wedge C? \wedge C! \\
C!e & \mathrel{\widehat{=}} (\neg C? \wedge C! \wedge \mathsf{stable}\,(C)\ ;\ \mathsf{skip}) \vee \mathsf{empty}\ ;\ C.e \\
C?x & \mathrel{\widehat{=}} (\neg C! \wedge C? \wedge \mathsf{stable}\,(C)\ ;\ \mathsf{skip}) \vee \mathsf{empty}\ ;\ C.x
\end{array}$$

Π_i is the projection function that for $i = 1$ gives the "willing to read" value, for $i = 2$ gives the "willing to write" value and for $i = 3$ the actual value in the channel. In the first interval the agent is waiting for its partner and in the second interval communication takes place.

Let $d \in TIME$. The notation $C!_d e$ $(C?_d x)$ specifies an agent which is willing to perform the communication at time d. However, the agent will be held up forever if the environment fails to react promptly.

$$\begin{array}{l}
C!_d e \mathrel{\widehat{=}} C!e \wedge (\mathit{finite} \supset \mathit{len} = d) \\
C?_d x \mathrel{\widehat{=}} C?x \wedge (\mathit{finite} \supset \mathit{len} = d)
\end{array}$$

4.2 Shunt communication

Let s be a shunt then $\mathsf{shunt}\,s\ \mathsf{in}\ P$ denotes that a new shunt s is introduced. $\mathsf{write}\,(v, s)$ denotes that value v is written to shunt s, $\mathsf{read}\,(s)$ gives the value stored in shunt s and $\surd s$ gives the stamp of shunt s. These agents are defined as followed

$$\begin{array}{ll}
\surd s & \mathrel{\widehat{=}} \Pi_1(s) \\
\mathsf{shunt}\,s\ \mathsf{in}\ P & \mathrel{\widehat{=}} \exists s \bullet \surd s = 0 \wedge P \\
\mathsf{write}\,(v, s) & \mathrel{\widehat{=}} \mathsf{skip} \wedge \bigcirc s = (\surd s + 1, v) \\
\mathsf{read}\,(s) & \mathrel{\widehat{=}} \Pi_2(s)
\end{array}$$

where Π_i is the projection function that for $i = 1$ gives the stamp and for $i = 2$ gives the value stored in the shunt.

Let $d \in Time - \{0\}$. The notation $\mathsf{write}_d(v, s)$ specifies an agent that writes value v to shunt s at time d.

$$\mathsf{write}_d(v, s) \mathrel{\widehat{=}} len = d - 1 \,;\, \mathsf{skip} \wedge \bigcirc s = (\surd s + 1, v)$$

Note: the value of stamp is defined to be the value of the stamp in the previous state plus 1.

If one wants a version of write_d which remains stable except of the last state of the interval one can take $\mathsf{pwrite}_d(v, s)$ which is defined as

$$\mathsf{pwrite}_d(v, s) \mathrel{\widehat{=}} \mathsf{write}_d(v, s) \wedge padded(s)$$

4.3 Delay and timeout

Let $d \in TIME \cup \{\infty\}$. The notation delay_d describes an agent which first holds up for d time units and then terminates with all global variables untouched. Its execution does not claim any resource time

$$\mathsf{delay}_d \mathrel{\widehat{=}} len = d$$

Let $d \in TIME \cup \{\infty\}$. The notation $P \trianglelefteq_d Q$ defines an agent which behaves like P if P is executed within d time units, otherwise it behaves like agent Q.

$$P \trianglelefteq_d Q \mathrel{\widehat{=}} \mathsf{if}\,(P \supset finite \wedge len \leq d)\ \mathsf{then}\ P\ \mathsf{else}\ Q$$

4.4 Resource allocation

Let res be a resource then $\mathsf{request}\,(v, res)$ is the agent that requests v units of resource res. If these v units are not available it waits for them. $\mathsf{release}\,(v, res)$ is the agent that releases v units of resource res.

$$\mathsf{request}\,(v, res) \mathrel{\widehat{=}} \mathsf{if}\ res \geq v\ \mathsf{then}\ res := res - v\ \mathsf{else}\ \bigcirc(\mathsf{request}\,(v, res))$$
$$\mathsf{release}\,(v, res) \mathrel{\widehat{=}} \bigcirc res = res + v$$

4.5 Parallel Composition

Let P and Q be infinite agents their parallel composition $P \parallel Q$ behaves as if P and Q are running independently except that all the communications between P and Q have to be synchronized, that is, whenever one agent outputs a message on any link between the two agents, another simultaneously inputs that message from the same channel. Formally it can be defined by

$$P \parallel Q \mathrel{\widehat{=}} (P \wedge Q)$$

If P or Q are not infinite agents then a skeleton must be devised that deals with their termination properly, i.e., the first terminating agent has to wait on the other one only then the parallel construct terminates. For P and Q finite agents:

$$\begin{aligned} P \parallel Q \mathrel{\widehat{=}} \exists t_1, t_2 \bullet\ & \\ & ((more \supset t_1 = 0) \wedge P \wedge padded\,(t_1)\,;\, t_1 = 1 \wedge halt\,(t_2 = 1)) \wedge \\ & ((more \supset t_2 = 0) \wedge Q \wedge padded\,(t_2)\,;\, t_2 = 1 \wedge halt\,(t_1 = 1)) \end{aligned}$$

4.6 The funnel

The funnel agent performs a restricted form of multiplexing on shunts. The syntax is defined as $s_i \rightsquigarrow_I s_{out}$ to describe the connection of shunts s_i (indexed by I) to the shunt s_{out}, so that if any write occurs in s_j, ($j \in I$) then it occurs at the same time in s_{out}. If no write occurs to any of s_i then s_{out} remains stable, and if two different values are written to s_i and s_j at the same time then the funnel becomes *false*.

$$
\begin{aligned}
s_i \rightsquigarrow_I s_v \mathrel{\widehat{=}} & (\textstyle\bigwedge_{i \in I} \mathsf{stable}\,(s_i) \wedge \mathsf{stable}\,(s_{out})) \vee \\
& ((\textstyle\bigvee_{i \in I} \mathsf{stable}\,(s_i)\,;\mathsf{skip} \wedge \surd s_i := \surd s_i + 1) \wedge \\
& \exists v, t \bullet \mathit{len} = t \wedge \\
& (\textstyle\bigwedge_{i \in I} \mathsf{stable}\,(s_i)\,;\mathsf{skip} \wedge \surd s_i := \surd s_i + 1 \supset \mathit{fin}(\mathsf{read}\,(s_i)) = v) \\
& \wedge \mathsf{pwrite}_t(v, s_{out}))
\end{aligned}
$$

The intuition behind this funnel is that it allows concurrently executing agents to write to the same shunt, assuming no conflict may occur. Note that this behavior cannot be described without a new construct because an agent which performs reading and writing to shunts requires at least two time units (because of updating of the stamp), and the funnel is instantaneous.

The funnel agent is not performing a global state change instantaneously, instead it is using a specific characteristic of a given system (that of "agreement" between shunt writers). In order to define a new agent. The funnel still requires at least one time unit in which to execute if it is to perform any writing at all, but this time is logically shared with the agents which are actually performing the write. The funnel can be thought as the association of a single physical shunt by a number of logical ones.

5 Algebraic laws

In this section we explore some of the algebraic properties of the agents introduced into ITL. But first, we define the *refinement* ordering relation $\sqsubseteq$ in the normal way as:

$$P \sqsubseteq Q \mathrel{\widehat{=}} Q \supset P$$

Clearly, $\sqsubseteq$ is a partial order. As usual then,

- a sequence $\{P_k\}$ of agents is called increasing if P_k gets progressively stronger

  $$\forall k \bullet P_k \sqsubseteq P_{k+1}$$

- a sequence $\{P_k\}$ of agents is called decreasing if P_k gets progressively weaker

  $$\forall k \bullet P_{k+1} \sqsubseteq P_k$$

It is elementary fact that each

- increasing sequence has a limit which is the weakest agent stronger than each P_k (namely $\bigcap_k P_k$).

– decreasing sequence has a limit which is the strongest agent weaker than each P_k (namely $\bigcup_k P_k$).

Let $\mathcal{P}$ be the set of all agents. A function F from $\mathcal{P}$ to $\mathcal{P}$

– is $\wedge$-continuous if for every increasing chain $\{P_k\}$: $F(\bigcap_k P_k) = \bigcap_k F(P_k)$,
– is $\vee$-continuous if for every decreasing chain $\{P_k\}$: $F(\bigcup_k P_k) = \bigcup_k F(P_k)$ and
– is monotonic if $P \sqsubseteq Q \supset F(P) \sqsubseteq F(Q)$.

5.1 Non-deterministic choice

Let P and Q be two agents, $P \vee Q$ denotes an agent that behaves either as P or Q, but does not determine which one. Hence the environment cannot control or predict the result. The following are some basic laws governing $\vee$.
The choice between the same agents is vacuous.

$(\vee -1)\ P \vee P \quad \equiv \quad P$

The choice is commutative and associative

$(\vee -2)\ P \vee Q \quad \equiv \quad Q \vee P$
$(\vee -3)\ P \vee (Q \vee R) \quad \equiv \quad (P \vee Q) \vee R$

The choice has *true* as its zero

$(\vee -4)\ \mathit{true} \vee P \quad \equiv \quad \mathit{true}$

Note: for agent P and Q we have $P \sqsubseteq Q \quad \equiv \quad (P \vee Q) \equiv P$

5.2 If then else–conditional

The conditional is both idempotent and associative

$(\mathsf{if} -1)\ \mathsf{if}\ f_0\ \mathsf{then}\ f\ \mathsf{else}\ f \quad \equiv \quad f$
$(\mathsf{if} -2)\ \mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ (\mathsf{if}\ f_0\ \mathsf{then}\ f_2\ \mathsf{else}\ f_3)$
$\equiv \quad \mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_3$
$\equiv \quad \mathsf{if}\ f_0\ \mathsf{then}\ (\mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2)\ \mathsf{else}\ f_3$

The following two laws describe how conditional makes a choice between its arguments.

$(\mathsf{if} -3)\ \mathsf{if}\ \mathit{true}\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2 \quad \equiv \quad f_1$
$(\mathsf{if} -4)\ \mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2 \quad \equiv \quad \mathsf{if}\ \neg f_0\ \mathsf{then}\ f_2\ \mathsf{else}\ f_1$

The relationship between conditional and $\vee$ is given by

$(\mathsf{if} -5)\ \mathsf{if}\ f_0\ \mathsf{then}\ (f_1 \vee f_2)\ \mathsf{else}\ f_3 \quad \equiv \quad (\mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_3) \vee (\mathsf{if}\ f_0\ \mathsf{then}\ f_2\ \mathsf{else}\ f_3)$
$(\mathsf{if} -6)\ (\mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2) \vee f_3 \quad \equiv \quad \mathsf{if}\ f_0\ \mathsf{then}\ (f_1 \vee f_3)\ \mathsf{else}\ (f_2 \vee f_3)$

To allow us unnesting of conditionals we have

$(\mathsf{if} -7)\ \mathsf{if}\ f_{00}\ \mathsf{then}\ (\mathsf{if}\ f_{01}\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2)\ \mathsf{else}\ (\mathsf{if}\ f_{02}\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2)$
$\equiv \quad \mathsf{if}\,(\mathsf{if}\ f_{00}\ \mathsf{then}\ f_{01}\ \mathsf{else}\ f_{02})\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2$

5.3 Chop–sequential composition

The following rules describes the characteristics of ;.
; has empty as a unit and is associative

$$(;-1)\ \mathsf{empty}\,;f \quad\equiv\quad f \quad\equiv\quad f\,;\mathsf{empty}$$
$$(;-2)\ (f_1\,;f_2)\,;f_3 \quad\equiv\quad f_1\,;(f_2\,;f_3)$$

The chop operator distributes over nondeterministic choice and conditional

$$(;-3)\ f_1\,;(f_2\vee f_3)\,;f_4 \quad\equiv\quad (f_1\,;f_2\,;f_4)\vee(f_1\,;f_3\,;f_4)$$
$$(;-4)\ (\mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2)\,;f_3 \quad\equiv\quad \mathsf{if}\ f_0\ \mathsf{then}\ (f_1\,;f_3)\ \mathsf{else}\ (f_2\,;f_3)$$

The chop operator is $\vee$-continuous

$$(;-5)\ f_1\,;\textstyle\bigcup_i g_i \quad\equiv\quad \bigcup_i f_1\,;g_i$$

5.4 While loop

The while loop $\mathsf{while}\ f_0\ \mathsf{do}\ f_1$ behaves like the sequential composition of f_1 and $\mathsf{while}\ f_0\ \mathsf{do}\ f_1$ if f_0 is true otherwise it terminates.

$$(\mathsf{while}-0)\ \mathsf{while}\ f_0\ \mathsf{do}\ f_1 \quad\equiv\quad ((f_0\wedge f_1)\,;\mathsf{while}\ f_0\ \mathsf{do}\ f_1)\vee(\neg f_0\wedge\mathsf{empty})$$

Note: if f_0 is a formula without temporal operators then $(\mathsf{while}-0)$ simplifies to the usual

$$(\mathsf{while}-1)\ \mathsf{while}\ f_0\ \mathsf{do}\ f_1 \quad\equiv\quad \mathsf{if}\ f_0\ \mathsf{then}\ f_1\,;(\mathsf{while}\ f_0\ \mathsf{do}\ f_1)\ \mathsf{else\ empty}$$

We will assume that f_0 is a formula without temporal operators in the following exposition.
In general we are interested in the solution of

$$X \quad\equiv\quad \mathsf{if}\ f_0\ \mathsf{then}\ (f_1\,;X)\ \mathsf{else\ empty}$$

The strongest solution is $\mathsf{while}\ f_0\ \mathsf{do}\ f_1$.

$$(\mathsf{while}-2)\ \text{If}\ X\sqsubseteq(\mathsf{if}\ f_0\ \mathsf{then}\ (f_1\,;X)\ \mathsf{else\ empty})\ \text{then}\ X\sqsubseteq\mathsf{while}\ f_0\ \mathsf{do}\ f_1$$

Now, let

$$P_0 \quad\equiv\quad \mathit{false}$$
$$P_{n+1} \quad\equiv\quad \mathsf{if}\ f_0\ \mathsf{then}\ (P\,;P_n)\ \mathsf{else\ empty}$$

it is clear that $\forall n\bullet P_{n+1}\sqsubseteq P_n$. Consequently, the sequence $\{P_n\}$ is decreasing, and by taking n large enough, we can approximate as closely as we wish to the behavior of $\mathsf{while}\ f_0\ \mathsf{do}\ P$. The loop $\mathsf{while}\ f_0\ \mathsf{do}\ P$ can be defined as the limit of all approximations.

$$(\mathsf{while}-3)\ \mathsf{while}\ f_0\ \mathsf{do}\ P \quad\equiv\quad \textstyle\bigcup_k P_k$$

where $\bigcup$ is the least upper bound.

$$(\mathsf{while}-4)\ (\mathsf{while}\ f_{01}\ \mathsf{do}\ P)\,;(\mathsf{while}\ f_{01}\vee f_{02}\ \mathsf{do}\ P)\equiv\mathsf{while}\ f_{01}\vee f_{02}\ \mathsf{do}\ P$$

This law helps in proving the correctness of ;.

5.5 Delay

The following are some laws for the delay agent:

$$
\begin{array}{lcl}
(\mathsf{delay}-1)\ \mathsf{delay}_{d_1}\ ;\ \mathsf{delay}_{d_2} & \equiv & \mathsf{delay}_{d_1+d_2} \\
(\mathsf{delay}-2)\ \mathsf{delay}_{d_0} \trianglelefteq_{d_1} Q & \equiv & \mathsf{if}\ d_0 \leq d_1\ \mathsf{then}\ \mathsf{delay}_{d_0}\ \mathsf{else}\ Q \\
(\mathsf{delay}-3)\ \mathsf{skip} & \equiv & \mathsf{delay}_1
\end{array}
$$

5.6 Parallel

The following are some laws for the parallel agent.

$$
\begin{array}{lcl}
(\parallel -1)\ P \parallel \mathit{true} & \equiv & P \\
(\parallel -2)\ P \parallel Q & \equiv & Q \parallel P \\
(\parallel -3)\ P \parallel (Q_1 \vee Q_2) & \equiv & (P \parallel Q_1) \vee (P \parallel Q_2) \\
(\parallel -4)\ (P \parallel Q) \parallel R & \equiv & P \parallel (Q \parallel R) \\
(\parallel -5)\ (\mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2) \parallel f_3 & \equiv & \mathsf{if}\ f_0\ \mathsf{then}\ (f_1 \parallel f_3)\ \mathsf{else}\ (f_2 \parallel f_3)
\end{array}
$$

6 Application

In this section we illustrate the use of some of the previous concrete agents on the well known example of the Watchdog Timer. We adopt the following requirement. We assume a finite number of shunts d_i with some indexing set I, which are written to by dog processes. Associated with each d_i is a period p_i, we expect at least one write to occur to each d_i every p_i. If a write fails to occur on one or more d_i then a boolean shunt labeled Alm is written within p_j (for the shortest period p_j for which a write failed). Each shunt d_i is monitored for a total of n_i periods.

The properties of this watchdog timer can be defined as a conjunction of properties of individual dog monitors as follows:

$$
\begin{array}{l}
\mathit{Prop}_i \mathrel{\widehat{=}} (\mathit{len} = p_i \wedge \\
\qquad \mathsf{stable}\,(d_i)\ \supset\ \mathsf{pwrite}_{p_i}(\mathit{true}, \mathit{alm}_i) \wedge \\
\qquad \neg\,\mathsf{stable}\,(d_i)\ \supset\ \mathsf{stable}\,(\mathit{alm}_i) \\
\qquad)^{n_i}\ ;\ \mathsf{stable}\,(\mathit{alm}_i)
\end{array}
$$

Along with the property Join that states that if an individual dog shunt produces alm, then the output channel Alm is written to, otherwise it remains stable.

$$
\begin{array}{l}
\mathit{Join} \mathrel{\widehat{=}} \\
(\bigwedge_i \mathsf{stable}\,(\mathit{alm}_i) \wedge \mathsf{stable}\,(\mathit{Alm})) \vee \\
(\bigvee_i \mathsf{stable}\,(\mathit{alm}_i)\ ;\ \mathsf{skip} \wedge \bigcirc\surd \mathit{alm}_i = \surd \mathit{alm}_i + 1) \wedge \\
\exists v, t \bullet \mathit{len} = t \wedge \\
(\bigwedge_i \mathsf{stable}\,(\mathit{alm}_i)\ ;\ \mathsf{skip} \wedge \bigcirc\surd \mathit{alm}_i = \surd \mathit{alm}_i + 1\ \supset\ \mathit{fin}(\mathsf{read}\,(\mathit{alm}_i)) = v) \\
\wedge\ \mathsf{pwrite}_t(v, \mathit{Alm}))
\end{array}
$$

Clearly the property $Join$ suggests a funnel on the output shunts alm_i. The specification for the watchdog timer can now be given by the agent:

$$\{Alm, \overline{d}\} : \exists \overline{alm} \bullet \bigwedge_{i \in I} Prop_i \wedge Join$$

We can see that the underlying architecture of the concrete agent which implements this specification will have the structure given in figure 1.

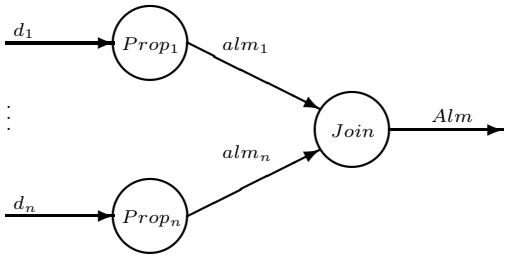

Fig. 1. Structure of the watchdog

The first refinement step is replacing the $\bigwedge$ by the $\|$ and introduction of shunts alm_i:

$$(*)\ \mathsf{shunt}\ \overline{alm}\ \ \mathsf{in}\ \|_{i \in I}\ \{alm_i, d_i\} : Prop_i \parallel \{Alm, \overline{alm}\} : Join$$

If one can find refinements S_i and S for respectively $Prop_i$ and $Join$ one can infer because of *compositionality* that

$$\mathsf{shunt}\ \overline{alm}\ \ \mathsf{in}\ \|_{i \in I}\ S_i \parallel S$$

is a refinement of $(*)$. The refinement of property $Prop_i$ is as follows

$$\begin{array}{l}
\{alm_i, d_i\} : Prop_i \\
\sqsubseteq \\
\mathsf{for}\ n_i\ \mathsf{times\ do}\ (len = p_i \wedge \\
\quad \mathsf{stable}\,(d_i)\ \supset\ \mathsf{pwrite}_{p_i}(true, alm_i) \wedge \\
\quad \neg\, \mathsf{stable}\,(d_i)\ \supset\ \mathsf{stable}\,(alm_i) \\
)\ ;\, \mathsf{stable}\,(alm_i) \\
\sqsubseteq \\
\mathsf{for}\ n_i\ \mathsf{times\ do}\ (len = p_i \wedge \\
\quad \mathsf{if}\ \surd d_i = fin(\surd d_i)\ \mathsf{then}\ \mathsf{pwrite}_{p_i}(true, alm_i)\ \mathsf{else}\ \mathsf{stable}\,(alm_i) \\
)\ ;\, \mathsf{stable}\,(alm_i)
\end{array}$$

This concrete agent is periodic with a period of p_i. At the start and ending of each period it reads the stamp of the dog shunt d_i. The two values are compared and if they are equal then the alm_i is written at the end of the period otherwise alm_i remains stable.

The refinement of property $Join$ is the funnel:

$$\{Alm, \overline{alm}\} : Join \sqsubseteq alm_i \rightsquigarrow_I Alm$$

Leaving us with the concrete watchdog agent:

$$
\begin{array}{l}
\mathsf{shunt}\ \overline{alm}\ \ \mathsf{in} \\
\|_{i \in I}\ \mathsf{for}\ n_i\ \mathsf{times\ do}\ (len = p_i \wedge \\
\qquad \mathsf{if}\ \surd d_i = fin(\surd d_i)\ \mathsf{then}\ \mathsf{pwrite}_{p_i}(true, alm_i)\ \mathsf{else}\ \mathsf{stable}\,(alm_i) \\
\qquad)\ ;\ \mathsf{stable}\,(alm_i) \\
\|\ alm_i \rightsquigarrow_I Alm
\end{array}
$$

7 Conclusion and future work

The current development technique of computing systems used by ITL and its associated executable subset, Tempura, is only suitable for small sequential Tempura programs. This paper introduces an alternative development technique for ITL based on the refinement calculus of [1,6]. In this technique one starts with an "abstract" specification and transforms this specification via correctness preserving refinement laws into a concrete specification. During this transformation abstract and concrete constructs are allowed to intermix. By enlarging the set of concrete constructs, the range of concrete programs (specifications) that can be developed in such a way has been extended.

Currently we are embarking on applying our technique on a large system, namely the development of an event processor known as EP/3 [2]. In addition, we are planing to incorporate our refinement calculus into PVS so as to have an integrated mechanical tool set that allows such development to be mechanically carried out and checked. The syntax, semantics and the proof system of ITL has already been incorporated in PVS [3]. By linking such a tool set with Tempura allows a complete development environment within which an ITL specification can be checked, analyzed, executed and then refined to a concrete implementation. Such an implementation may be then translated into a hardware/software implementation language. For example in the case of EP/3, the translation may be done to, e.g. Verilog, and subsequently a circuit diagram can be produced.

Acknowledgements

We would like to thank Ben Moszkowski for his comments and criticism. Furthermore the referees made many helpful comments.

References

1. R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988
2. A. Cau, H. Zedan, N. Coleman and B. Moszkowski. Using ITL and Tempura for Large Scale Specification and Simulation, in proc. of fourth euromicro workshop on parallel and distributed processing, IEEE, 1996, Braga, Portugal, 493–500.
3. A. Cau and B. Moszkowski: Using PVS for Interval Temporal Logic Proofs, Part 1: The Syntactic and Semantic Encoding. Technical Report, 1996.

4. J. He. A dual-time model for communicating sequential processes. Unpublished manuscript.
5. R. Milner. A calculus for communicating processes. LNCS 92, 1983.
6. C. Morgan. Programming from specifications. Prentice-Hall International, 1990.
7. B. Moszkowski: A Temporal Logic for Multilevel Reasoning About Hardware. IEEE Computer 1985;18:10-19.
8. B. Moszkowski: Executing Temporal Logic Programs. Cambridge Univ. Press, Cambridge, UK, 1986.
9. B. Moszkowski. Some very compositional temporal properties, in: Programming Concepts, Methods and Calculi, Ernst-Rüdiger Olderog (ed.), IFIP Transactions, Vol. A-56, North-Holland, 1994, 307-326.
10. X. Nicolin, J. Richier, J. Sifakis and J. Voiron. ATP: an algebra for timed processes. In Programming Concepts and Methods, M. Broy and C.B. Jones (eds), pp. 414–443, 1990.
11. D. Scholefield, H. Zedan and J. He. A specification oriented semantics for the refinement of real-time systems. *Theoretical Computer Science*, 130, August 1994.

A Some refinement laws

In this appendix we list some refinement laws.

Law B (Strengthen specification). *If* $f_1 \supset f_0$ *then* $w : f_0 \sqsubseteq w : f_1$.

Law C (New variable). $w : f \sqsubseteq \mathsf{var}\, v \;\mathsf{in}\; w \cup \{v\} : f$.

Law D (New channel). $w : f \sqsubseteq \mathsf{channel}\; C \;\mathsf{in}\; w \cup \{C\} : f$.

Law E (New shunt). $w : f \sqsubseteq \mathsf{shunt}\;\; s \;\mathsf{in}\; w \cup \{s\} : \surd s = 0 \wedge f$.

Law F (Parallel composition). *then*

$$w_0 \cap w_1 : f_0 \wedge f_1 \sqsubseteq (w_0 : f_0) \parallel (w_1 : f_1).$$