Using PVS for Interval Temporal Logic Proofs*
Part 1: The Syntactic and Semantic Encoding

Antonio Cau
Science and Engineering Research Centre
Department of Computer Science
De Montfort University
The Gateway
Leicester LE1 9BH, UK
E-mail: cau@dmu.ac.uk

Ben Moszkowski
Department of Electrical and Electronic Engineering
University of Newcastle upon Tyne
Newcastle NE1 7RU, UK
E-mail: Ben.Moszkowski@newcastle.ac.uk

12 May 2005

Abstract

Interval temporal logic (ITL) is a logic that is used to specify and reason about systems. The logic has a powerful proof system but rather than doing proofs by hand, which is tedious and error prone, we want a tool that can check each proof step. Instead of developing a new tool we will use the existing prototype verification system (PVS) as basic tool. The specification language of PVS is used to encode interval temporal logic semantically and syntactically. With this we can encode the ITL proof system within PVS. Several examples of proofs in ITL that are done per hand are checked with PVS.

1 Introduction

Interval temporal logic (ITL) is a very convenient formalism for the description of hardware and software systems [4]. It describes these systems in terms of intervals which are sequences of states wherein a systems can be. Also an executable subset of ITL has been defined the so called Tempura language. A system is first specified in this language and then this specification is "executed" by the Tempura simulator, i.e., it tries to construct the sequence states of the system corresponding to this specification. This simulator is a very helpful tool for constructing a specification for a system. The correctness, with respect to certain properties of, can not be shown by this simulator (although for very simple systems it is possible). The correctness of systems is therefore shown with help of a powerful proof system[5, 6]. Experience with this proof system shows that a whole range of properties can be proven. Currently ITL is used to specify and verify a general purpose multi-threaded data-flow processor EP/3[1].

One drawback is that all these proofs are done "by hand", i.e., there is no tool that checks that a particular application of a proof rule is right. For simple systems the proof task is still manageable but

*Funded by EPSRC Research Grant GR/K25922

Table 1: Syntax of ITL

Expressions	
$exp ::=$	$\mu \mid a \mid A \mid g(exp_1, \ldots, exp_n) \mid \imath a{:}\, f$

Formulas	
$f ::=$	$p(exp_1, \ldots, exp_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathsf{skip} \mid f_1 ; f_2 \mid f^*$

for complex systems, like the EP/3, it is nearly impossible. So we decided to construct a proof assistant for ITL. Rather than constructing it from scratch we took an existing proof tool and embed ITL within it. We took as proof tool the prototype verification system (PVS)[7] since it has an excellent reputation and it is easy to use. This proof tool was already used for the embedding of the duration calculus[8] which is a descendant of ITL. This embedding was a semantical one, an extra interface was constructed to deal with the syntax of the duration calculus. We didn't want to proceed this way because it means an extra interface to be built. So we tried to embed ITL semantically and syntactically within PVS.

In section 2 we give a brief introduction of ITL. In section 3 we discuss the embedding of ITL within PVS. In section 4 we give an evaluation and discuss related work.

2 ITL

We first give the syntax and semantics of ITL and then give some axioms and proof rules plus an example proof.

2.1 Syntax and semantics of ITL

An interval is considered to be a (in)finite sequence of states, where a state is a mapping from variables to their values. The length of an interval is equal to 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 μ is an integer value, 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.

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

- $\imath a{:}\, f$: the value of a such that f holds.
- $\forall v \bullet f$: for all v 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 f_1 holds for the whole interval.
- f^*: holds if the interval is decomposable into a (in)finite number of intervals such that for each of them 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.

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

Frequently used abbreviations are listed in table 2.

2.2 Proof system of ITL

First we discuss the propositional case and then the first order case. Rather than giving a full listing of the propositional axioms and proof rules we present some basic ones and give some example proofs.

ChopOrImp	$\vdash$	$f_0;(f_1 \vee f_2) \supset (f_0;f_1) \vee (f_0;f_2)$
BiBoxChopImpChop	$\vdash$	$\boxdot(f_0 \supset f_1) \wedge \Box(f_2 \supset f_3) \supset (f_0;f_2) \supset (f_1;f_3)$
MP	$\vdash$	$f_0 \supset f_1, \quad \vdash f_0 \quad \Rightarrow \quad \vdash f_1$
BoxGen	$\vdash$	$f_0 \quad \Rightarrow \quad \vdash \Box f_0$
BiGen	$\vdash$	$f_0 \quad \Rightarrow \quad \vdash \boxdot f_0$

We now give a few sample theorems and their proofs:

BoxChopImpChop $\vdash \quad \Box(f_0 \supset f_1) \supset (f_2;f_0) \supset (f_2;f_1)$

Table 2: Frequently used abbreviations

$true$	$\stackrel{\text{def}}{\equiv} 0 = 0$	true value
$f_1 \vee f_2$	$\stackrel{\text{def}}{\equiv} \neg(\neg f_1 \wedge \neg f_2)$	f_1 or f_2
$f_1 \supset f_2$	$\stackrel{\text{def}}{\equiv} \neg f_1 \vee f_2)$	f_1 implies f_2
$f_1 \quad \equiv \quad f_2$	$\stackrel{\text{def}}{\equiv} (f_1 \supset f_2) \wedge (f_1 \supset f_2)$	f_1 equivalent f_2
$\exists v \bullet f$	$\stackrel{\text{def}}{\equiv} \neg\forall v \bullet \neg f$	there exists a v s.t. f
$\bigcirc f$	$\stackrel{\text{def}}{\equiv} \mathsf{skip}\,; f$	next f
inf	$\stackrel{\text{def}}{\equiv} true\,; false$	infinite interval
$finite$	$\stackrel{\text{def}}{\equiv} \neg inf$	finite interval
$\Diamond f$	$\stackrel{\text{def}}{\equiv} finite\,; f$	(sometimes f)
$\Box f$	$\stackrel{\text{def}}{\equiv} \neg\Diamond\neg f$	always f
$\odot f$	$\stackrel{\text{def}}{\equiv} \neg\bigcirc\neg f$	weak next f
$\diamondsuit f$	$\stackrel{\text{def}}{\equiv} f\,; true$	some initial subinterval
$\boxdot f$	$\stackrel{\text{def}}{\equiv} \neg(\diamondsuit\neg f)$	all initial subintervals
if f_0 then f_1 else f_2	$\stackrel{\text{def}}{\equiv} (f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$	if then else
$\bigcirc exp$	$\stackrel{\text{def}}{\equiv} \iota a{:}\bigcirc(exp = a)$	next value
$fin\ exp$	$\stackrel{\text{def}}{\equiv} \iota a{:}fin\,(exp = a)$	end value
$A := exp$	$\stackrel{\text{def}}{\equiv} \bigcirc A = exp$	assignment
$more$	$\stackrel{\text{def}}{\equiv} \bigcirc true$	non-empty interval
empty	$\stackrel{\text{def}}{\equiv} \neg more$	empty interval
$fin\ f$	$\stackrel{\text{def}}{\equiv} \Box(\mathsf{empty} \supset f)$	final state
while f_0 do f_1	$\stackrel{\text{def}}{\equiv} (f_0 \wedge f_1)^* \wedge fin\ \neg f_0$	while loop

Proof:

1	$f_2 \supset f_2$	Prop
2	$\boxdot(f_2 \supset f_2)$	1, BiGen
3	$\boxdot(f_2 \supset f_2) \wedge \Box(f_0 \supset f_1) \supset (f_2; f_0) \supset (f_2; f_1)$	BiBoxChopImpChop
4	$\Box(f_0 \supset f_1) \supset (f_2; f_0) \supset (f_2; f_1)$	2, 3, Prop

RightChopImpChop $\vdash f_0 \supset f_1 \quad \Rightarrow \quad \vdash f_2; f_0 \supset f_2; f_1$

Proof:

1	$f_0 \supset f_1$	given
2	$\Box(f_0 \supset f_1)$	BoxGen
3	$\Box(f_0 \supset f_1) \supset (f_2; f_0) \supset (f_2; f_1)$	BoxChopImpChop
4	$f_2; f_0 \supset f_2; f_1$	2, 3, MP

ChopOrEqv $\vdash \quad f_0; (f_1 \vee f_2) \quad \equiv \quad f_0; f_1 \vee f_0; f_2$

The proof for $\subset$ is immediate from axiom ChopOrImp. Here is the proof for the converse:

1	$f_1 \supset f_1 \vee f_2$	Prop
2	$f_0; f_1 \supset f_0; (f_1 \vee f_2)$	1, RightChopImpChop
3	$f_2 \supset f_1 \vee f_2$	Prop
4	$f_0; f_2 \supset f_0; (f_1 \vee f_2)$	3, RightChopImpChop
5	$f_0; f_1 \vee f_0; f_2 \supset f_0; (f_1 \vee f_2)$	2, 4, Prop

Some axioms for the first order case are shown below. We let v and v' refer to both static and location variables. We denote by f_v^e that in formula f expression e is substituted for variable v.

ForallElim	$\vdash \quad \forall v \bullet f \supset f_v^e$, where the expression e has the same data and temporal type as the variable v and is free for v in f.
ExistsChopRight	$\vdash \quad \exists v \bullet (f_1; f_2) \supset (\exists v \bullet f_1); f_2$, where v doesn't occur freely in f_2.
ForallGen	$\vdash \quad f \Rightarrow \vdash \quad \forall v \bullet f$, for any variable v.

3 Embedding of ITL within PVS

In this section we first give the syntactic embedding followed by the embeddings of the semantics and the proof system of ITL.

3.1 Syntactic encoding

In table 1 the syntactic definition of expressions is given. As such we can't encode this in PVS the problematic construct is $\iota a: f$. Before we can encode this, f (formulae) have to be encoded but for the latter we need the encoding of expressions. We have a chicken and egg problem here. But luckily this is not really a problem because the $\iota a: f$ construct is mainly used to encode the value of an expression in the next state and the value of an expression at the end of the interval. We will use $\bigcirc exp$ instead of $\iota a: \bigcirc(exp = a)$ and $fin\ exp$ instead of $\iota a: fin(exp = a)$. In table 1 we also use $g(exp_1, \ldots, exp_n)$ where g is a function instead of a general g we will only use the integer $+, -$ and $*$ functions.

The syntactic encoding of expressions is then as follows using the abstract data-type construct:

```
%%%% definition of datatype expression
exp : DATATYPE
  BEGIN
   const(n: int)                 : cst?       : exp
   svariable(sv: nat)            : svr?       : exp
   variable(v: nat)              : vr?        : exp
   nx(nxexp : exp)               : nx?        : exp
   fin(finexp : exp)             : fin?       : exp
   +(aex1: exp, aex2: exp)       : plus?      : exp
   -(sex1: exp, sex2: exp)       : min?       : exp
   *(mex1: exp, mex2: exp)       : mult?      : exp
  END exp
```

The *sv* and *v* are the identifier of respectively a static and a state variable. Since these identifiers are natural numbers we have an unbounded number of static and state variables. This is exactly what we want. With subtype declaration like `z: (vr?)` we can express that `z` ranges over the state variables.

For the encoding of formulae we need the encoding of expressions so this is imported. Again, in table 1 we used a general relation p on expressions, we will only use $<$ and $=$ relations. The syntactic encoding is then as follows:

```
%%%% definition of datatype formula
form : DATATYPE
 BEGIN
  importing exp
  FA  (v1: (vr?  ), f1: form)      : fa?       : form
  FAs (v3: (svr? ), f1fas : form)  : fas?      : form
  skip                             : skip?     : form
  =(eqxp1: exp, eqxp2: exp)        : eqi?      : form
  <(lexp1: exp, lexp2: exp)        : les?      : form
  -(fn1: form)                     : inot?     : form
  /\(fa1: form, fa2: form)         : iand?     : form
  ^(fc1: form, fc2: form)          : chop?     : form
  chopstar(fcs1: form)             : chopstar? : form
 END form
```

The abbreviations listed in table 2 can now be encoded as follows:

```
%%%% frequently used abbreviations
  T               : form = (const(0)=const(0));
  \/(f1,f2)       : form = -((-f1) /\ (-f2));
  =>(f1,f2)       : form = (-f1) \/ f2;
  ==(f1,f2)       : form = (f1 => f2) /\ (f2 => f1);
  TE(va,f1)       : form = -FA(va,-f1);
  TEs(sva,f1)     : form = -FAs(sva,-f1);
  O(f1)           : form = skip^f1;
  inf             : form = T^F;
  finite          : form = -inf;
  <>(f1)          : form = finite^f1;
  [](f1)          : form = -(<>(-f1));
  wO(f1)          : form = -(O(-f1));
  Di(f1)          : form = f1^T;
  Bi(f1)          : form = -(Di(-f1));
  ife(f0, f1, f2) : form = (f0 /\ f1) \/ (-f0 /\ f2);
  as(exp1,exp2)   : form = nx(exp1) = exp2;
  more            : form = O(T);
  empty           : form = -more;
  fin(f1)         : form = [](empty => f1);
  while(f0, f1)   : form = chopstar(f0 /\ f1) /\ fin(-f0)
```

3.2 Semantic encoding

Before we can give the semantics of the above syntactic constructs we must define intervals (i.e., (in)finite sequences of states). First we will encode (in)finite sequences. We denote a(n) (in)finite sequence as a record of three fields, the first field is a boolean indicating if the sequence is infinite, the second field is a natural number indicating the length of the sequence and the third field is an array whose indices are bounded if the sequence is finite. For the encoding of f^* and $\forall v \cdot f$ we need definitions of respectively sequences of natural numbers and sequences of values. So we will give a general definition of sequences in that the sequence elements are of general type T.

```
%%%% definition of an (in)finite sequence
  Sequ: TYPE =
```

```
              [# infinite : bool,
                 len: nat,
                 seq: ARRAY[{i:nat | infinite or i<=len} -> T] #]
```

The infinite and finite subtypes are defined as follows:

```
%%%% definition of an infinite sequence
  Infsequ: TYPE = { tau0: Sequ | infinite(tau0) }

%%%% definition of a finite sequence
  Finsequ: TYPE = { tau0: Sequ | not infinite(tau0) }
```

We also define the notions of subsequence and suffix of a sequence. They are straight forward:

```
%%%% sequ is the same as Sequ
  sequ: TYPE = Sequ

  tau : VAR sequence

%%%% definition of subsequence
 sub(tau0: sequ,
     m0: {i:nat | infinite(tau0) or i<=len(tau0)},
     n0: {i:nat | m0<=i AND (infinite(tau0) OR i<=len(tau0))})
  : Finsequ =
   LET lsum = n0-m0 IN
   (# infinite:=false,
      len :=  lsum,
      seq := (LAMBDA (x: {i:nat|i<=lsum} ) : seq(tau0)(x + m0))
   #)

%%%% definition of suffix of a sequence
  suf(tau0 : Infsequ, m0 : nat) : Infsequ =
    (# infinite:= infinite(tau0),
       len := len(tau0),
       seq := (LAMBDA (x: {i:nat|true}): seq(tau0)(x + m0))
    #)
```

Next is the definition of state. In section 2.1 a state was a mapping from both the static and state variables to their values. In PVS, however, we will have two kinds of states; one is a mapping from static variables to their values and the other one is a mapping from state variables to their values. This makes reasoning about them easier in PVS. The variables are identified by a natural number and the values are just integers. So the encoding is as follows:

```
%%%% state variables are of the sort integers
  Value: TYPE = int

%%%% static variables are of the sort integers
  SValue: TYPE = int

%%%% Vars the indices of state variables (infinite number)
  Vars: TYPE = nat

%%%% SVars the indices of static variables (infinite number)
```

```
  SVars: TYPE = nat

%%%% State(i): the ith state variable

  State: TYPE = [Vars -> Value]

%%%% SState(i): the ith static variable
  SState: TYPE = [SVars -> SValue]
```

Now we are able to define the semantics of the syntactic constructs. Since we have split the state the semantics is a little bit different than in section 2.1. Instead of interpreting over sequences of states, we will interpret over a pair $(env, sigma)$ where *env* is a mapping form static variables to their values and *sigma* is a sequence of mappings from state variables to their values. With this we enforce that the static variables don't change in an interval because they do not depend on intervals.

First the semantics of expressions. This is mapping from the syntactic constructs to integer values. Since we defined expressions recursively we give a denotational semantics. This is straight forward, only the semantics of $\bigcirc exp$ and *fin exp* is interesting. But first the uninteresting part:

We need a definition of sequence of states, the following imports the general theory for sequences and instantiate it for states.

```
%%%% importing the theory of sequences instantiated for states
importing sequ[State]

  Interval    : TYPE = sequ[State]

  env            : VAR SState
  sigma          : VAR Interval
%%%% semantics of expression
  E(e : exp)(env,sigma) : RECURSIVE Value =
     CASES e OF
     const(v)         : v,
     variable(n)      : seq(sigma)(0)(n),
     svariable(n)     : env(n),
     nx(exp1)         : semnx(E(exp1))(env,sigma),
     fin(exp1)        : semfin(E(exp1))(env,sigma),
     +(exp1, exp2)    : E(exp1)(env,sigma) + E(exp2)(env,sigma),
     -(exp1, exp2)    : E(exp1)(env,sigma) - E(exp2)(env,sigma),
     *(exp1, exp2)    : E(exp1)(env,sigma) * E(exp2)(env,sigma)
     ENDCASES
   MEASURE sizeexp(e)
```

The semantics of $\bigcirc exp$ is problematic because it is undefined for intervals of length 0. How to encode that an expression has an undefined value? If we look at the semantics given in section 2.1 we see that we use there the choice operator, i.e., an undefined value is just any value! In PVS there is also such a construct: it is the epsilon construct. The semantics of $\bigcirc exp$ and *fin exp* (undefined for infinite intervals) are as follows:

```
  IValue       : TYPE = [SState,Interval -> Value]
  E1             : VAR IValue;
%%%% semantics of O(expression)
  semnx(E1)(env,sigma) : Value =
   epsilon(lambda x1 : if infinite(sigma) then
                         E1(env,suf(sigma,1))=x1 elsif
                          len(sigma)>0 then
```

```
                        E1(env,sub(sigma,1,len(sigma)))=x1
                         else false endif)

%%%% semantics of fin(expression)
  semfin(E1)(env,sigma) : Value =
   epsilon(lambda x1 : if infinite(sigma) then
                           false else
                       E1(env,sub(sigma,len(sigma),len(sigma)))=x1
                        endif)
```

If one uses recursion in PVS one has to give a function so that can be determined that the "definition" terminates. In this case this function (the length of an expression) is as follows:

```
%%%% lenght of an expression (needed for recursive definition)
  sizeexp(e:exp) : nat =
     reduce_nat(
     (LAMBDA (i : int): 1 + abs(i) ),  %const(n)
     (LAMBDA (i : nat): 1 + i ),       %svariable(sv)
     (LAMBDA (i : nat): 1 + i ),       %variable(v)
     (LAMBDA (i : nat) : 1 + i),       %nx(exp1)
     (LAMBDA (i : nat) : 1 + i),       %fin(exp2)
     (LAMBDA (i, j : nat): 1 + i + j), %+(exp1, exp2)
     (LAMBDA (i, j : nat): 1 + i + j), %-(exp1, exp2)
     (LAMBDA (i, j : nat): 1 + i + j)  %*(exp1, exp2)
      )(e)
```

The semantics of formulae is a bit more complicated as seen in section 2.1. Especially the $\forall v \cdot f$ and f^* constructs. The rest is straight forward as seen below: The semantics of a formula is a mapping from the syntactic constructs to the boolean values.

```
  Iform        : TYPE = [SState,Interval -> bool]
  F1,F2          : VAR Iform;
%%%% semantics of -f
 semnot(F1)(env,sigma) : bool = not F1(env,sigma)

%%%% semantics of f1 /\ f2
 semand(F1,F2)(env,sigma) : bool = F1(env,sigma) and F2(env,sigma)

%%%% semantics of f1^f2
 semchop(F1,F2)(env,sigma) : bool =
   (EXISTS (m: nat):
      ( (infinite(sigma) and F2(env,suf(sigma,m))) or
        (not infinite(sigma) and m <= len(sigma) and
          F2(env,sub(sigma, m, len(sigma)))
        )
      )
      and F1(env,sub(sigma, 0, m))
   )
   or (infinite(sigma) and F1(env,sigma) )

%%%% semantics of formulae
  M(f:form)(env,sigma) : RECURSIVE bool =
     CASES f OF
       FA(v,f1)     : semforall(M(f1),v)(env,sigma),
       FAs(v,f1)    : semsforall(M(f1),v)(env,sigma),
```

```
      skip         : (len(sigma) = 1 and not infinite(sigma)),
      -(f1)        : semnot(M(f1))(env,sigma),
      =(exp1,exp2) : E(exp1)(env,sigma) = E(exp2)(env,sigma),
      <(exp1,exp2) : E(exp1)(env,sigma) < E(exp2)(env,sigma),
      ^(f1,f2)     : semchop(M(f1),M(f2))(env,sigma),
      /\(f1,f2)    : semand(M(f1),M(f2))(env,sigma),
      chopstar(f1) : semchopstar(M(f1))(env,sigma)
    ENDCASES
  MEASURE sizeform(f)
```

We first discuss the semantics of f^*. As seen in section 2.1 we need a (in)finite list of chopping points in an interval. These chopping points are natural numbers. Since we already defined (in)finite sequences of any type we can use that to define this list of chopping points. The semantics of f^* is now straight forward as shown below:

```
%%%% importing theory of sequences instantiated for natural numbers
importing sequ[nat]

%%%% definition of infinite list of chopping points
  ininterval   : TYPE = Infsequ[nat]

%%%% definiton of finite list of chopping points
  fninterval   : TYPE = Finsequ[nat]

  il             : VAR ininterval
  fl             : VAR fninterval

%%%% semantics of chopstar(f)
  semchopstar(F1)(env,sigma) : bool =
    (IF infinite(sigma) THEN
      (EXISTS fl :
         seq(fl)(0) = 0 and
         F1(env,suf(sigma, seq(fl)(len(fl)))) and
         (FORALL (i: below[len(fl)]):
             seq(fl)(i) < seq(fl)(i + 1) and
             F1(env,sub(sigma, seq(fl)(i), seq(fl)(i + 1)))
         )
      )
      OR
      (EXISTS il:
         seq(il)(0) = 0 and
         (FORALL (i: nat):
            seq(il)(i) < seq(il)(i + 1) and
            F1(env,sub(sigma, seq(il)(i), seq(il)(i + 1)))
         )
      )
     ELSE
      (EXISTS fl :
         seq(fl)(0) = 0 and
         seq(fl)(len(fl)) = len(sigma) and
         (FORALL (i: below[len(fl)]):
            seq(fl)(i+1) <= len(sigma) and
            seq(fl)(i) < seq(fl)(i + 1) and
            F1(env,sub(sigma, seq(fl)(i), seq(fl)(i + 1)))
         )
```

```
     )
    ENDIF)
```

At last we discuss the semantics of $\forall v \cdot f$. Because we have split the state the semantics of $\forall v \cdot f$ is also split into two cases. The first and easy case is if v is a static variable. The semantics is as follows:

```
semsforall(F1:Iform,sva)(env,sigma) : bool =
   (FORALL x1 : F1(env with [(sv(sva)) := x1],sigma) )
```

The second case is if v is a state variable. As seen in section 2.1 we have to encode the $\sigma \sim_v \sigma'$ relation that denotes that *sigma* and σ' are the same except for the behavior of v. Instead of encoding this relation directly in PVS we encode this in a similar way as the static case. In the latter case the semantics of $\forall v \cdot f$ was encoded as for all values assigned to v, f should hold. The analogon for the state case is that for all values assigned to v (in the interval), f should hold. For this we need a (in)finite sequence of values. Because we have this type already defined the semantics is then as follows:

```
%%%% importing theory of sequences instantiated for values
importing sequ[Value]

%%%% definition of infinite list of values
  InfIIValue  : TYPE = Infsequ[Value]

%%%% definition of finite list of values
  FinIIValue  : TYPE = Finsequ[Value]

  ival           : VAR InfIIValue
  fval           : VAR FinIIValue

%%%% semantics of FA(va,f)
semforall(F1,va)(env,sigma) : bool =
   if infinite(sigma) then
    (FORALL ival :
      F1(env,
         (# infinite:=infinite(sigma),
            len:=len(sigma),
            seq:=(lambda (i: {j:nat|true}) :
                  seq(sigma)(i) with [(v(va)):=seq(ival)(i)]) #)))
   else
    (FORALL fval : len(fval)=len(sigma) implies
      F1(env,
         (# infinite:=infinite(sigma),
            len:=len(sigma),
            seq:=(lambda (i : {j:nat|j<=len(sigma)}) :
                  seq(sigma)(i) with [(v(va)):=seq(fval)(i)]) #)))
   endif
```

3.3 Proof system encoding

The propositional axioms and rules presented in section 2.2 are encoded as follows (note `V(f)` is a predicate that denotes that `f` holds for all intervals and interpretations of static intervals; this is needed in order to express the rules):

```
%%%% definition of validity of formulae
V: pred[form] =
  (LAMBDA f1: (FORALL env: (FORALL sigma: M(f1)(env,sigma))))
```

```
   CONVERSION V

%%%% the axioms
ChopOrImp: LEMMA  (f0^(f1 \/ f2)) => ((f0^f1) \/ (f0^f2))

BiBoxChopImpChop: LEMMA
  (Bi(f0 => f1) /\ [](f2 => f3)) => ((f0^f2) => (f1^f3))

%%%% the rules
MP: LEMMA V(f0 => f1) AND V(f0) IMPLIES V(f1)

BoxGen: LEMMA V(f0) IMPLIES V([](f0))

BiGen: LEMMA V(f0) IMPLIES V(Bi(f0))
```

The following example is a PVS proof session of the second proof in section 2.2:

- This is what we should prove:

```
RightChopImpChop :

  |-------
{1} (FORALL (f0: form, f1: form, f2: form):
      ((V(((f0 => f1))))) IMPLIES (V(((f2 ^ f0) => (f2 ^ f1)))))
```

- With skolimization we eliminate the for all quantor.

```
Rule? (SKOSIMP)
Skolemizing and flattening,
this simplifies to:
RightChopImpChop :

{-1}    ((V(((f0!1 => f1!1)))))
  |-------
{1}    (V(((f2!1 ^ f0!1) => (f2!1 ^ f1!1))))
```

- Apply proof rule `BoxGen`.

```
Rule? (FORWARD-CHAIN "BoxGen")
Forward chaining on BoxGen,
this simplifies to:
RightChopImpChop :

{-1}    V([](((f0!1 => f1!1))))
[-2]    ((V(((f0!1 => f1!1)))))
  |-------
[1]    (V(((f2!1 ^ f0!1) => (f2!1 ^ f1!1))))
```

- Add an instance of lemma `BoxChopImpChop`.

```
Rule?
(LEMMA "BoxChopImpChop" ("f0" "f0!1" "f1" "f1!1" "f2" "f2!1"))
```

```
Applying BoxChopImpChop where
  f0 gets f0!1,
  f1 gets f1!1,
  f2 gets f2!1,
this simplifies to:
RightChopImpChop :

{-1}    V([]((f0!1 => f1!1)) => (f2!1 ^ f0!1) => (f2!1 ^ f1!1))
[-2]    V([](((f0!1 => f1!1))))
[-3]    ((V(((f0!1 => f1!1)))))
  |-------
[1]    (V(((f2!1 ^ f0!1) => (f2!1 ^ f1!1))))
```

- Apply proof rule MP.

```
Rule? (FORWARD-CHAIN "MP")
Forward chaining on MP,
Q.E.D.


Run time  = 2.67 secs.
Real time = 16.34 secs.
```

The example shows that the PVS proof follows the same pattern as the "by hand" proof.

The encoding of the first order axioms is a bit more complicated because one has to encode "f_v^e" (substitution), "where the expression e has the same data and temporal type as the variable v and is free for v in f" and "where v doesn't occur freely in f_2". The latter is easy to encode. Assume v is a state variable (the static case is analogous). Because the syntax of expression and formulae are encoded as abstract datatypes the following will do the job:

```
%%%% set of free state variables in an expresion
freeexp(e:exp) : RECURSIVE setof[(vr?)] =
    CASES e OF
     const(n)          : emptyset,
     variable(v)       : singleton(variable(v)),
     svariable(sv)     : emptyset,
     nx(exp1)          : freeexp(exp1),
     fin(exp1)         : freeexp(exp1),
     +(aexp1, aexp2)   : union(freeexp(aexp1), freeexp(aexp2)),
     -(sexp1, sexp2)   : union(freeexp(sexp1), freeexp(sexp2)),
     *(mexp1, mexp2)   : union(freeexp(mexp1), freeexp(mexp2))
     ENDCASES
   MEASURE sizeexp(e)

%%%% set of free state variables in a formula
freeform(f:form) : RECURSIVE setof[(vr?)] =
    CASES f OF
     FA(v1,f1)       : difference(freeform(f1), singleton(v1)),
     FAs(v3,f1)      : freeform(f1),
     skip            : emptyset,
     =(eqxp1,eqxp2)  : union(freeexp(eqxp1), freeexp(eqxp2)),
     <(lexp1,lexp2)  : union(freeexp(lexp1), freeexp(lexp2)),
     -(f1)           : freeform(f1),
     ^(f1,f2)        : union(freeform(f1), freeform(f2)),
```

```
     /\(f1,f2)       : union(freeform(f1), freeform(f2)),
     chopstar(f1)    : freeform(f1)
     ENDCASES
   MEASURE sizeform(f)
```

Encoding of substitution is also relatively easy. In order to assure that expression *e* has the "same temporal datatype" as variable *v* when *e* is substituted for *v* we take the convention that *e* is an expression that contains no temporal operators, i.e., no ◯ and *fin* operators. This kind of subtype can be encoded as follows:

```
%%%% definition of expressions containing no temporal constructs
  pexp         : TYPE =
     { e: exp | forall (exp1: exp) :
                  not subterm(nx(exp1), e) and
                  not subterm(fin(exp1),e)}
```

Substitution is then encoded as follows:

```
%%%% definition of syntactic substitution of state variable by
%%%% an expression containing no temporal constructs
su(exp1, x, pexp2) : RECURSIVE exp =
    CASES exp1 OF
     const(n)         : const(n),
     variable(v)      : if variable(v) = x then
                        pexp2 else variable(v) endif,
     svariable(sv)    : svariable(sv),
     nx(nxexp1)       : nx(su(nxexp1,x,pexp2)),
     fin(finexp1)     : fin(su(finexp1,x,pexp2)),
     +(aexp1, aexp2)  : su(aexp1,x,pexp2) + su(aexp2,x,pexp2),
     -(sexp1, sexp2)  : su(sexp1,x,pexp2) - su(sexp2,x,pexp2),
     *(mexp1, mexp2)  : su(mexp1,x,pexp2) * su(mexp2,x,pexp2)
     ENDCASES
   MEASURE sizeexp(exp1)

%%%% definition of syntactic substitution of a state variable by
%%%% an expression containing no temporal constructs
suform(f1, x, pexp2) : RECURSIVE form =
 CASES f1 OF
     FA(v1,f1)       : FA(v1,suform(f1,x,pexp2)),
     FAs(v3,f1)      : FAs(v3,suform(f1,x,pexp2)),
     skip            : skip,
     =(eqxp1,eqxp2)  : su(eqxp1,x,pexp2) = su(eqxp2,x,pexp2),
     <(lexp1,lexp2)  : su(lexp1,x,pexp2) < su(lexp2,x,pexp2),
     -(f1)           : -(suform(f1,x,pexp2)),
     ^(f1,f2)        : suform(f1,x,pexp2)^suform(f2,x,pexp2),
     /\(f1,f2)       : suform(f1,x,pexp2) /\ suform(f2,x,pexp2),
     chopstar(f1)    : chopstar(suform(f1,x,pexp2))
     ENDCASES
   MEASURE sizeform(f1)
```

If one defines substitution has to take care that variables occurring in the substited expression doesn't become bound. To check that one can define functions that on expressions and formulae that give the bound variables. These functions are defined analogous as the "free variables" functions.

The encoding of the first order axioms of section 2.2 is the as follows:

```
%%%%%%%% first order
ForallSub: LEMMA not member(v1,bound(f1)) and
  (forall (z:(vr?)):
     member(z,freeexp(pexp1)) implies not member(z,bound(f1))) and
  (forall (z:(svr?)):
     member(z,sfreeexp(pexp1)) implies not member(z,sbound(f1)))
  implies
       V(FA(v1,f1) => suform(f1,v1,pexp1))

ExistsChopRight : LEMMA not member(v1,freeform(f2)) implies
          V(TE(v1,f1^f2) => (TE(v1,f1)^f2))

% the rules
ForallGen: LEMMA V(f0) implies V(FA(v1,f0))
```

4 Conclusion

We have used the ITL proof assistant to verify a list of more than 100 theorems. Experience shows that proofs within the tool almost follow the pattern as the "by hand" case. This ensures that people who are used to the proofs by hand can easily switch to the proofs by the tool. The next step will be the verification of a large example. This example will be the EP/3 example[1] for which already an ITL specification exist (a large ITL formula of about 3500 lines). This example will probably require the definition of proof strategies (tactics). These strategies can be defined in PVS to semi-automatically prove certain theorems. Besides proof strategies also compositional proof rules are needed to tackle the EP/3 example. These proof rules are discussed in [5, 6]. Because the basic ITL formalism is now encoded the encoding of these proof rules is straight forward.

For ITL we also defined constructs to reason about message-passing communication and timing (delay and time-out) using the work on temporal agent model (TAM) [9]. The latter has also a refinement calculus which can be easily ported to ITL[2]. These constructs and refinement rules will also be included in the proof assistant.

Related work is done in Macau where Mao Xiaoguang, Xu Qiwen and Wang Ji are working on a proof assistant for interval logics. They have embedded the neighbourhood calculus within PVS[3]. This calculus can express a whole range of interval logics like the duration calculus and ITL. We have exchanged ideas, they have, on our suggestion, also used abstract datatypes to syntactically encode their calculus. We didn't want to follow their idea of one general interval logic but merely wanted a practical proof assistant for checking ITL proofs.

References

[1] 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.

[2] A. Cau and H. Zedan. Communication and Time in ITL, in preparation.

[3] X. Mao, Q. Xu and J. Wang. Towards a proof assistant for interval logics, in preparation.

[4] B. Moszkowski. Executing temporal logic programs, Cambridge Univ. Press, UK, 1986.

[5] 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.

[6] B. Moszkowski. Using temporal fixpoints to compositionally reason about liveness, in proc. of the 7th BCS FACS Refinement Workshop, He Jifeng (ed.), Bath, UK, 1996.

[7] John Rushby. A tutorial on specification and verification using PVS. In proc. of the First International Symposium of Formal Methods Europe FME '93: Industrial-Strength Formal Methods, Peter Gorm Larsen (ed.), 1993, Odense, Denmark, 357–406. Check home-page: `http://www.csl.sri.com/pvs.html`

[8] J.U. Skakkebæk and N. Shankar. Towards a Duration Calculus Proof Assistant in PVS, in proc. of the 3rd International Symposium Formal Techniques in Real-Time and Fault-Tolerant Systems FTRTFT '94, Hans Langmaack, Willem-Paul de Roever and Jan Vytopil (eds.), 1994, Lübeck, Germany, 660–679.

[9] D. Scholefield, H. Zedan and J. He. A specification oriented semantics for the refinement of real-time systems. *Theoretical Computer Science*, 130, August 1994.