Interval Temporal Logic Proof Checker[1]
——DRAFT:——

Antonio Cau
Software Technology Research Laboratory
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

April 14, 1997

[1]Funded by EPSRC Research Grant GR/K25922: A Compositional Approach to the Specification of Systems using ITL and Tempura.
See http://www.cms.dmu.ac.uk/~cau/project.html for information on how to get the PVS libraries of the ITL proof checker.

Contents

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, checked with the PVS system, are given.

Chapter 1

Introduction

Interval temporal logic (ITL) is a very convenient formalism for the description of hardware and software systems [5]. 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 of 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, 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 the proof system[6, 7] of ITL. 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 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)[8] since it has an excellent reputation and it is easy to use. PVS is an interactive environment, developed at SRI, for writing formal specifications and checking formal proofs. The specification language used in PVS is a strongly typed higher order logic. This specification language is powerful enough for specifying the syntax, semantics of ITL, and the proof system of ITL. The powerful interactive theorem prover/proof checker of PVS has a large set of basic deductive steps and the facility to combine these steps into proof strategies. This proof tool was already used for the embedding of the duration calculus[10] which is a descendant of ITL. This embedding was a semantical one, an extra external 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 5 we give the conclusion and discuss future work.

Chapter 2

Interval temporal logic

We first give the syntax and semantics of ITL and then give some axioms and proof rules plus an example proof which is later checked with the PVS system.

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 2.1 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:

- $\iota a\colon f$: 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.

Table 2.1: Syntax of ITL

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

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.

- $\mathcal{E}_\sigma[\![v]\!] = \sigma_0(v)$.
- $\mathcal{E}_\sigma[\![g(exp_1,\ldots,exp_n)]\!] = \hat{g}(\mathcal{E}_\sigma[\![exp_1]\!],\ldots,\mathcal{E}_\sigma[\![exp_n]\!])$.
- $\mathcal{E}_\sigma[\![\iota a{:}\, f]\!] = \begin{cases} \chi(u) & \text{if } u \neq \{\} \\ \chi(a) & \text{otherwise} \end{cases}$
  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 $\hat{p}(\mathcal{E}_\sigma[\![exp_1]\!],\ldots,\mathcal{E}_\sigma[\![exp_n]\!])$.
- $\mathcal{M}_\sigma[\![\neg f]\!] = \text{tt}$ 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 $\mathcal{M}_\sigma[\![f_2]\!] = \text{tt}$.
- $\mathcal{M}_\sigma[\![\forall v \bullet f]\!] = \text{tt}$ 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 2.2–2.5.

Table 2.2: Frequently used non-temporal abbreviations

$true$	$\widehat{=} 0 = 0$	true value
$false$	$\widehat{=} \neg true$	false value
$f_1 \vee f_2$	$\widehat{=} \neg(\neg f_1 \wedge \neg f_2)$	or
$f_1 \supset f_2$	$\widehat{=} \neg f_1 \vee f_2$	implies
$f_1 \equiv f_2$	$\widehat{=} (f_1 \supset f_2) \wedge (f_1 \supset f_2)$	equivalent
$\exists v \bullet f$	$\widehat{=} \neg \forall v \bullet \neg f$	exists

Table 2.3: Frequently used temporal abbreviations

$\bigcirc f$	$\widehat{=}$ skip $; f$	next
$more$	$\widehat{=} \bigcirc true$	non-empty interval
empty	$\widehat{=} \neg more$	empty interval
inf	$\widehat{=} true ; false$	infinite interval
$isinf(f)$	$\widehat{=} inf \wedge f$	is infinite
$finite$	$\widehat{=} \neg inf$	finite interval
$isfin(f)$	$\widehat{=} finite \wedge f$	is finite
$fmore$	$\widehat{=} more \wedge finite$	non-empty finite interval
$\Diamond f$	$\widehat{=} finite ; f$	sometimes
$\Box f$	$\widehat{=} \neg\Diamond\neg f$	always
$\textcircled{w} f$	$\widehat{=} \neg\bigcirc\neg f$	weak next
$\textcircled{i} f$	$\widehat{=} f ; true$	some initial subinterval
$\boxed{i} f$	$\widehat{=} \neg(\textcircled{i}\neg f)$	all initial subintervals
$\textcircled{d} f$	$\widehat{=} finite ; f ; true$	some subinterval
$\boxed{a} f$	$\widehat{=} \neg(\textcircled{d}\neg f)$	all subintervals
$\textcircled{fin} f$	$\widehat{=} \Diamond(more \wedge f)$	
$\boxed{fin} f$	$\widehat{=} \neg(\textcircled{fin}\neg f)$	

Table 2.4: Frequently used concrete abbreviations

if f_0 then f_1 else f_2	$\widehat{=}\ (f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$	if then else
if f_0 then f_1	$\widehat{=}$ if f_0 then f_1 else empty	if then
fin f	$\widehat{=}\ \Box(\text{empty} \supset f)$	final state
sfin f	$\widehat{=}\ \neg(\mathit{fin}(\neg f))$	strong final state
halt f	$\widehat{=}\ \Box(\text{empty} \equiv f)$	terminate interval when
shalt f	$\widehat{=}\ \neg(\mathit{halt}(\neg f))$	strong terminate interval when
keep f	$\widehat{=}\ \boxdot(\text{skip} \supset f)$	all unit subintervals
keepnow f	$\widehat{=}\ \diamondsuit(\text{skip} \wedge f)$	initial unit subinterval
f^{ω}	$\widehat{=}\ \mathit{isinf}(\mathit{isfin}(f)^*)$	infinite chopstar
fstar (f)	$\widehat{=}\ \mathit{isfin}(\mathit{isfin}(f)^*) \vee$	
	$\widehat{=}\ \mathit{isfin}(\mathit{isfin}(f)^*)\,;\mathit{isinf}(f)$	finite chopstar
while f_0 do f_1	$\widehat{=}\ (f_0 \wedge f_1)^* \wedge \mathit{fin}\,\neg f_0$	while loop
repeat f_0 until f_1	$\widehat{=}\ f_0\,;(\text{while } \neg f_1 \text{ do } f_0)$	repeat loop

Table 2.5: Frequently used abbreviations related to expressions

$\bigcirc exp$	$\widehat{=}\ \iota a{:}\bigcirc(exp = a)$	next value
fin exp	$\widehat{=}\ \iota a{:}\mathit{fin}\,(exp = a)$	end value
$A := exp$	$\widehat{=}\ \bigcirc A = exp$	assignment
$exp_1 \approx exp_2$	$\widehat{=}\ \Box(exp_1 = exp_2)$	equal in interval
$exp_1 \leftarrow exp_2$	$\widehat{=}\ \mathit{finite} \wedge (\mathit{fin}\ exp_1) = exp_2$	temporal assignment
exp_1 *gets* exp_2	$\widehat{=}\ \mathit{keep}\,(exp_1 \leftarrow exp_2)$	gets
stable exp	$\widehat{=}\ exp$ *gets* exp	stability
padded exp	$\widehat{=}\ \exists a \bullet \mathit{keep}\,(exp = a)$	padded expression
$exp_1 <\!\sim exp_2$	$\widehat{=}\ (exp_1 \leftarrow exp_2) \wedge \mathit{padded}\ exp_1$	padded temporal assignment
goodindex exp	$\widehat{=}\ \mathit{keep}\,(exp \leftarrow exp \vee exp \leftarrow exp + 1)$	goodindex
intlen (exp)	$\widehat{=}\ \exists I \bullet (I = 0) \wedge (I\ \mathit{gets}\ I + 1) \wedge \mathit{fin}\,(I = exp)$	interval length

2.2 Proof system of ITL

First we discuss the propositional case and then the first order case.

2.2.1 Propositional proof system for ITL

In table 2.6 we list the propositional axioms and rules for ITL.

We now give a few sample theorems and their proofs:

BiChopImpChop $\vdash \quad \boxdot(f_0 \supset f_1) \supset (f_0; f_2) \supset (f_1; f_2)$

Table 2.6: Propositional Axioms and Rules for ITL.

ChopAssoc	$\vdash$	$(f_0; f_1); f_2 \equiv f_0; (f_1; f_2)$
OrChopImp	$\vdash$	$(f_0 \vee f_1); f_2 \supset (f_0; f_2) \vee (f_1; f_2)$
ChopOrImp	$\vdash$	$f_0; (f_1 \vee f_2) \supset (f_0; f_1) \vee (f_0; f_2)$
EmptyChop	$\vdash$	$\mathsf{empty}; f_1 \equiv f_1$
ChopEmpty	$\vdash$	$f_1; \mathsf{empty} \equiv f_1$
BiBoxChopImpChop	$\vdash$	$\boxdot(f_0 \supset f_1) \wedge \Box(f_2 \supset f_3) \supset (f_0; f_2) \supset (f_1; f_3)$
StateImpBi	$\vdash$	$p \supset \boxdot p$
NextImpNotNextNot	$\vdash$	$\bigcirc f_0 \supset \neg\bigcirc\neg f_0$
KeepnowImpNotKeepnowNot	$\vdash$	$\mathit{keepnow}(f_0) \supset \neg \mathit{keepnow}(\neg f_0)$
BoxInduct	$\vdash$	$f_0 \wedge \Box(f_0 \supset \circledcirc f_0) \supset \Box f_0$
InfChop	$\vdash$	$(f_0 \wedge \mathit{inf}); f_1 \equiv (f_0 \wedge \mathit{inf})$
ChopStarEqv	$\vdash$	$f_0^* \equiv (\mathsf{empty} \vee ((f_0 \wedge \mathit{more}); f_0^*))$
ChopstarInduct	$\vdash$	$(\mathit{inf} \wedge f_0 \wedge \Box(f_0 \supset (f_1 \wedge \mathit{fmore}); f_0)) \supset f_1^*$
MP	$\vdash$	$f_0 \supset f_1, \vdash f_0 \Rightarrow \vdash f_1$
BoxGen	$\vdash$	$f_0 \Rightarrow \vdash \Box f_0$
BiGen	$\vdash$	$f_0 \Rightarrow \vdash \boxdot f_0$

Proof:

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

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

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

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

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

Proof:

1	$f_0 \supset f_1$	given
2	$\boxdot_i(f_0 \supset f_1)$	BiGen
3	$\boxdot_i(f_0 \supset f_1) \supset (f_0; f_2) \supset (f_0; f_2)$	BiChopImpChop
4	$f_0; f_2 \supset f_1; f_2$	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

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

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

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

2.2.2 First order proof system for ITL

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

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

Chapter 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 2.1 the syntactic definition of expressions is given. As such we can't encode this in PVS the problematic construct is $\imath a{:}\, f$. 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 $\imath a{:}\, f$ 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 $\imath a{:}\bigcirc(exp = a)$ and $fin\ exp$ instead of $\imath a{:}fin\,(exp = a)$. In table 2.1 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
   cst(n: int)                                  : cst?  : exp
   vr(v: nat, t: vrtype)                        : vr?   : exp
   ivr(iv: nat, it: ivrtype, iexp: exp)         : ivr?  : exp
   one(oe: exp, ot: onetype)                    : one?  : exp
   two(te1,te2: exp, tt: twotype)               : two?  : exp
   four(fe1,fe2,fe3,fe4: exp, ft: fourtype)     : four? : exp
  END exp
```

where

- `cst(n:int)` denotes integer constant n.
- `vr(v: nat, t: vrtype)` denotes a variable with name v and of type t. If $t = state$ then it denotes a state variable and if $t = static$ then it denotes a static variable. Since v ranges over the 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 variables.

- `ivr(iv: nat, it: ivrtype, iexp: exp)` denotes an array variable, i.e., *iv* is the name of the array and *it* is the type of the array and *iexp* is the index expression.
- `one(oe: exp, ot: onetype)` denotes the one place functions, if $ot = tnext$ then it denotes the "next" function and if $ot = tfin$ it denotes the "fin" function.
- `two(te1,te2: exp, tt: twotype)` denotes the two place functions, if $tt = tplus$ then it denotes the "+" function and if $tt = tmin$ then it denotes the "-" function and if $tt = ttimes$ then it denotes the "$*$" function and if $tt = tdiv$ then it denotes the "div" function and if $tt = tmod$ then it denotes the "mod" function.
- `four(fe1,fe2,fe3,fe4: exp, ft: fourtype)` denotes the four place functions, if $ft = teqce$ then it denotes the conditional expression "if fe1=fe2 then fe3 else fe4" and if $ft = tlece$ then it denotes the conditional expression "if fe1<fe2 then fe3 else fe4".

For the encoding of formulae we need the encoding of expressions so the abstract datatype of expressions is imported. Again, in table 2.1 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(va: (vr?), fa  : form)           : FA?       : form
  skip                                : skip?     : form
  etwo(ee1,ee2: exp, et: etwotype)    : etwo?     : form
  fone(fo: form, fot: fonetype)       : fone?     : form
  ftwo(ft1,ft2: form, ftt: ftwotype)  : ftwo?     : form
 END form
```

where

- `FA(va: (vr?), fa : form)` denotes "forall" quantification.
- `skip` denotes the 2 state interval, i.e., skip.
- `etwo(ee1,ee2: exp, et: etwotype)` denotes the 2 place predicate, if $et = teq$ then it denotes $ee1 = ee2$ and if $et = tle$ then it denotes $ee1 < ee2$.
- `fone(fo: form, fot: fonetype)` denotes the one place formulae, if $fot = tnot$ then it denotes $\neg fo$ and if $fot = tcs$ then it denotes fo^*.
- `ftwo(ft1,ft2: form, ftt: ftwotype)` denotes the 2 place formulae, if $ftt = tand$ then it denotes $ft1 \wedge ft2$ and if $ftt = tor$ then it denotes $ft1 \vee ft2$ and if $ftt = timpl$ then it denotes $ft1 \supset ft2$ and if $ftt = tchop$ then it denotes $ft1; ft2$.

The abbreviations listed in table 2.2–2.5 can now be encoded as follows:

```
%%%% frequently used abbreviations
  =(e1,e2)          : form = etwo(e1,e2,teq);
  <(e1,e2)          : form = etwo(e1,e2,tle);
  ifeq(e1,e2,e3,e4) : exp  = four(e1,e2,e3,e4,teqce);
  ifle(e1,e2,e3,e4) : exp  = four(e1,e2,e3,e4,tlece);
  -(f1)             : form = fone(f1,tnot);
  chopstar(f1)      : form = fone(f1,tcs);
  /\(f1,f2)         : form = ftwo(f1,f2,tand);
  \/(f1,f2)         : form = ftwo(f1,f2,tor);
  =>(f1,f2)         : form = ftwo(f1,f2,timpl);
  ^(f1,f2)          : form = ftwo(f1,f2,tchop);
  TE(va,f1)         : form = -FA(va,-f1);
  T                 : pform = (cst(0)=cst(0));
  F                 : pform = -T;
  O(f1)             : form = skip^f1;
  ==(f1,f2)         : form = (f1 => f2) /\ (f2 => f1);
  +(e1,e2)          : exp = two(e1,e2,tplus);
  -(e1,e2)          : exp = two(e1,e2,tmin);
  *(e1,e2)          : exp = two(e1,e2,ttimes);
  div(e1,e2)        : exp = two(e1,e2,tdiv);
  mod(e1,e2)        : exp = two(e1,e2,tmod);
  <=(e1,e2)         : form = (e1 < e2) \/ (e1 = e2);
  >(e1,e2)          : form = (e2 < e1);
  >=(e1,e2)         : form = (e1 > e2) \/ (e1 = e2);
  /=(e1,e2)         : form = -(e1=e2);
  more              : form = O(T);
  empty             : form = -more;
  inf               : form = T^F
  finite            : form = -inf
  fmore             : form = more /\ finite
  isinf(f1)         : form = f1 /\ inf
  isfin(f1)         : form = f1 /\ finite
  <>(f1)            : form = finite^f1;
  [](f1)            : form = -(<>(-f1));
  wO(f1)            : form = -(O(-f1));
  |>(f1,f2)         : form = -(f1^(-f2));
  Di(f1)            : form = f1^T
  Bi(f1)            : form = -(Di(-f1))
  Da(f1)            : form = finite^f1^T
  Ba(f1)            : form = -(Da(-f1))
  Dm(f1)            : form = <>(more /\ f1)
  Bm(f1)            : form = -(Dm(-f1))
  ife(f0, f1, f2)   : form = (f0 /\ f1) \/ (-f0 /\ f2)
  ifo(f0, f1)       : form = ife(f0, f1, empty)
  fin(f1)           : form = [](empty => f1)
  sfin(f1)          : form = -(fin(-f1))
  halt(f1)          : form = [](empty == f1)
```

```
  shalt(f1)          : form = -(halt(-f1))
  keep(f1)           : form = Ba(skip => f1)
  keepn(f1)          : form = Di(skip /\ f1)
  nx(e1)             : exp  = one(e1,tnext);
  fin(e1)            : exp  = one(e1,tfin);
  as(e1,e2)          : form = nx(e1) = e2
  equal(e1,e2)       : form = [](e1 = e2)
  tas(e1, e2)        : form = finite /\ (fin(e1) = e2)
  gets(e1, e2)       : form = keep(tas(e1, e2))
  stable(e1)         : form = gets(e1,e1)
  padded(e1)         : form = TE(sv0, keep(e1 = sv0))
  pta(e1, e2)        : form = tas(e1, e2) /\ padded(e1)
  goodindex(e1)      : form = keep(tas(e1,e1) \/ tas(e1,e1+cst(1)))
  intlen(e1)         : form = TE(v0, (v0 = cst(0)) /\
                                     gets(v0,v0+cst(1)) /\ fin(v0=e1))
  omega(f1)          : form = isinf(chopstar(isfin(f1)))
  fstar(f1)          : form = (isfin(chopstar(isfin(f1))) \/
                               isfin(chopstar(isfin(f1)))^isinf(f1))
  while(f0, f1)      : form = chopstar(f0 /\ f1) /\ fin(-f0)
  repeat(f0, f1)     : form = f0^while(-f1, 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 will denote an (in)finite sequence by 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 of bounded or unbounded length depending on whether the sequence is finite or not. For the encoding of f^* and $\forall v \bullet f$ 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. We also define the notions of subsequence and suffix of a sequence. They are straight forward.

```
%%%% definition of an (in)finite sequence
  sequ : TYPE = [# infinite : bool, len : nat,
                   seq : ARRAY[{i:nat | infinite or i<=len} -> T] #]
%%%% set of indexi of a sequence
  index(tau0 : sequ) : TYPE = {i:nat | infinite(tau0) or i<=len(tau0)}
%%%% set of indexi starting from m0 of a sequence
  index2(tau0 : sequ, m0 : index(tau0)) : TYPE =
    {i:nat |  m0 <= i and (infinite(tau0) or i <= len(tau0)) }
%%%% definition of subsequence
  sub(tau0 : sequ, m0 : index(tau0), n0 : index2(tau0,m0)) : sequ =
   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 : sequ, m0 : index(tau0)) : sequ =
   if infinite(tau0) then
    (# infinite := infinite(tau0), len := len(tau0),
       seq := (lambda (x: {i:nat|   true}): seq(tau0)(x + m0)) #)
   else let lsum = len(tau0)-m0 in
    (# infinite := infinite(tau0), len := lsum,
       seq := (lambda (x: {i:nat|i<=lsum}): seq(tau0)(x + m0)) #)
   endif
```

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 state variables to their values and the other one is a mapping from static variables to their values. The variables are identified by a natural number and the values are just integers. Since we also allow "indexed" variables we have to indicate how to interpret them. They are interpret over "indexed" states, i.e., a mapping from the Cartesian product of variables and values (of the indexi) to the values. So the encoding is as follows:

```
%%%% variables are of the sort integers
  Value: TYPE = int
%%%% names of state/static variables (infinite number)
   Vars: TYPE = nat
  SVars: TYPE = nat
%%%%  State of state/static variable
  State: TYPE  = [ [Vars -> Value] , [Vars,Value -> Value] ]
  SState: TYPE = [ [SVars -> Value] , [SVars,Value -> Value]  ]
```

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 from that of section 2.1. Instead of interpreting over sequences of states, we will interpret over a pair $(env, sigma)$ where *env* is a mapping from 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. 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. We will use this choice construct in the semantics of $\bigcirc exp$, *fin exp*, $div(exp1, exp2)$ and $mod(exp1, exp2)$ below.

```
importing sequ[State]

divs : LIBRARY = "/home/charity/pvs2+/lib/bitvectors"
importing divs@div, divs@mod

%%%% semantics of normal variables
  varsem(i,j1,env,sigma) : Value =
   if j1=state
```

```
   then PROJ_1(seq(sigma)(0))(i)
   else PROJ_1(env)(i)
   endif

%%%% semantics of indexed variables
  ivarsem(i,j1,x1,env,sigma) : Value =
   if j1=state
   then PROJ_2(seq(sigma)(0))(i,x1)
   else PROJ_2(env)(i,x1)
   endif

%%%% semantics of next and fin
  onesem(E1,j1,env,sigma) : Value =
  if j1=tnext
  then if infinite(sigma) or 1<=len(sigma)
       then E1(env,suf(sigma,1))
       else epsilon(lambda x1 : false)
       endif
  else if infinite(sigma)
       then epsilon(lambda x1 : false)
       else E1(env,suf(sigma,len(sigma)))
       endif
  endif

%%%% semantics of +,-,*,div and mod
  twosem(x1,x2,j4) : Value =
   if j4=tplus
   then x1 + x2
   elsif j4=tmin
   then x1 - x2
   elsif j4=ttimes
   then x1 * x2
   elsif j4=tdiv
   then if x2 /=0 then div(x1,x2)
        else epsilon(lambda x :  false )
        endif
   else if x2 /=0 then mod(x1,x2)
        else epsilon(lambda x1 :  false)
        endif
   endif

%%%% semantics of conditional expressions
  foursem(x1,x2,x3,x4,j1) : Value =
   if j1=teqce
   then if x1=x2 then x3 else x4 endif
   else if x1<x2 then x3 else x4 endif
   endif
```

```
%%%% semantics of expression
  E(e : exp)(env,sigma) : RECURSIVE Value =
    CASES e OF
     cst(n)            : n,
     vr(v,t)           : varsem(v,t,env,sigma),
     ivr(iv,it,iexp)   : ivarsem(iv,it,E(iexp)(env,sigma),env,sigma),
     one(oe,ot)        : onesem(E(oe),ot,env,sigma),
     two(te1,te2,tt)   : twosem(E(te1)(env,sigma),E(te2)(env,sigma),tt),
     four(fe1,fe2,fe3,fe4,ft): foursem(E(fe1)(env,sigma),
                                       E(fe2)(env,sigma),
                                       E(fe3)(env,sigma),
                                       E(fe4)(env,sigma), ft)
     ENDCASES
   MEASURE sizeexp(e)
```

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 ),                          %cst(n)
    (LAMBDA (i:nat, j:upto(1)): 1+i),              %vr(v,t)
    (LAMBDA (i:nat, j:upto(1), k:nat): 1+i+k),     %ivr(iv,t,exp)
    (LAMBDA (i:nat, j:upto(1)) : 1+i),             %one(e1,ot)
    (LAMBDA (i,j:nat, k:upto(4)): 1+i+j),          %two(e1,e2,tt)
    (LAMBDA (i,j,k,l:nat, m:upto(1)): 1+i+j+k+l)   %four(e1,e2,e3,e4,ft)
   )(e)
```

The semantics of formulae is a bit more complicated as seen in section 2.1. Especially the $\forall v \bullet f$ and f^* constructs. The rest is straight forward as seen below. We first discuss the semantics of f^*. 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 then straight forward as shown below. Note this is a compact version, if one does a case analysis it corresponds with the one in section 2.1.

The semantics of $\forall v \bullet f$ is split into two cases. The first 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 (second) case. In the latter case the semantics of $\forall v \bullet f$ is 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. The semantics of both is shown below.

```
importing sequ[nat], sequ[Value]

%%%% semantics of not and chopstar
  semfone(F1,j1,env,sigma) : bool =
```

```
  if j1=tnot then not F1(env,sigma)
  else (len(sigma)=0 and not infinite(sigma)) or
       (EXISTS l : seq(l)(0) = 0 and
         ( (not infinite(l) and
            ( (infinite(sigma) or seq(l)(len(l)) < len(sigma)) and
              F1(env,suf(sigma, seq(l)(len(l)))) or
            )
           ) or
           (infinite(l) and infinite(sigma))
         ) and
         (FORALL (i: nat): i<len(l) or infinite(l) implies
             (infinite(l) or infinite(sigma) or seq(l)(i+1)<len(sigma)) and
             seq(l)(i) < seq(l)(i + 1) and
             F1(env,sub(sigma, seq(l)(i), seq(l)(i + 1)))
         )
       )
  endif

%%%% semantics of and, or, implies and chop
  ^^(F1,F2) : Iform =  (lambda (env,sigma):
   (EXISTS (m: index[State](sigma)):
           F1(env,sub(sigma, 0, m)) and F2(env,suf(sigma,m))
   )
   or (infinite(sigma) and F1(env,sigma) ));


  semftwo(F1,F2,j3,env,sigma) : bool =
  if j3=tand     then F1(env,sigma) and F2(env,sigma)
  elsif j3=tor   then F1(env,sigma) or F2(env,sigma)
  elsif j3=timpl then F1(env,sigma) implies F2(env,sigma)
                 else (F1^^F2)(env,sigma)
  endif

%%%% semantics of forall quantification
  semforall(F1,va,env,sigma) : bool =
   if t(va)=state then
     (FORALL val : infinite(val)=infinite(sigma) and len(val)=len(sigma)
           implies
        F1(env,
           (# infinite:=infinite(sigma),
              len:=len(sigma),
              seq:=(lambda (i: index[State](sigma)) :
                       (PROJ_1(seq(sigma)(i)) with
                      [(v(va)):= seq(val)(i)], PROJ_2(seq(sigma)(i))) ) #)))
   else
    (FORALL x1 : F1( (PROJ_1(env) with
                [(v(va)) := x1],
```

```
                      PROJ_2(env)) ,sigma) )
   endif

%%%% semantics of predicates < and =
 semetwo(x1,x2,j1) : bool =
  if j1=teq then x1=x2  else x1<x2 endif

%%%% semantics of formulae
  M(f:form)(env,sigma) : RECURSIVE bool =
     CASES f OF
       FA(v,f1)          : semforall(M(f1),v,env,sigma),
       skip              : (len(sigma) = 1 and not infinite(sigma)),
       etwo(ee1,ee2,et)  : semetwo(E(ee1)(env,sigma),E(ee2)(env,sigma),et),
       fone(fo, fot)     : semfone(M(fo),fot,env,sigma),
       ftwo(ft1,ft2,ftt) : semftwo(M(ft1),M(ft2),ftt,env,sigma)
     ENDCASES
   MEASURE sizeform(f)
```

Below are lemmas that give the semantics of some derived formulae and expressions. Most of them can be proved automatically within PVS.

```
 l_more: LEMMA
   M(more)(env,sigma) =
     ((not infinite(sigma) and len(sigma) > 0) or infinite(sigma))

 l_inf: LEMMA
   M(inf)(env,sigma) = (infinite(sigma))

 l_always: LEMMA
   M([](f1))(env,sigma) =
     (FORALL (i: index[State](sigma)):  M(f1)(env,suf(sigma,i)))

 l_wnext1: LEMMA
        (Forall (i:index[State](sigma)): infinite(sigma) implies
          M(wO(f1))(env,suf(sigma, i))= M(f1)(env,suf(sigma, i + 1)))

 l_wnext2: LEMMA
        (Forall (i:index[State](sigma)):
           not infinite(sigma) IMPLIES
              M(wO(f1))(env,suf(sigma, i))
                   =
                (i = len(sigma)
                     OR
                   (i < len(sigma)
                        AND M(f1)(env,suf(sigma, 1 + i)))))


 l_nx_v_inf: LEMMA
```

```
   infinite(sigma)   implies
     E(nx(va))(env,sigma) =
      if t(va)=state then PROJ_1(seq(sigma)(1))(v(va))
      else PROJ_1(env)(v(va))
      endif

 l_nx_v_finite: LEMMA
   not infinite(sigma) implies
      E(nx(va))(env,sigma) =
        if len(sigma)=0 then epsilon(lambda (x1 : Value): false)
        else
          if t(va)=state then PROJ_1(seq(sigma)(1))(v(va))
          else PROJ_1(env)(v(va))
          endif
        endif

 l_fin: LEMMA
   not infinite(sigma) and t(va)=state implies
         M(fin(va=e1))(env,sigma) = (
          PROJ_1(seq(sigma)(len(sigma)))(v(va))=
          E(e1)(env,sub(sigma,len(sigma),len(sigma))))

 l_tas: LEMMA
   not infinite(sigma) and t(va)=state implies
           M(tas(va,e1))(env,sigma) =
            (PROJ_1(seq(sigma)(len(sigma)))(v(va))=E(e1)(env,sigma))
```

As a small exercise we have proved that the length of an interval can be expressed in ITL

```
 l_getsinc1: LEMMA
   not infinite(sigma) and t(va)=state implies
          M(gets(va,va+cst(1)))(env,sigma) =
           (forall (i:nat): i<=len(sigma) implies
            PROJ_1(seq(sigma)(i))(v(va))=PROJ_1(seq(sigma)(0))(v(va))+i)

%%%% length of an interval can be expressed
 l_intlen : LEMMA
   not infinite(sigma) implies
     (M(intlen(cst(x1)))(env,sigma) iff (len(sigma) = x1))
```

3.3 Proof system encoding

3.3.1 Propositional ITL proof system encoding

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

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

%%%% the axioms
ChopAssoc: LEMMA  |-((f0^f1)^f2) == (f0^(f1^f2))

OrChopImp: LEMMA  |-(((f0 \/ f1)^f2) => ((f0^f2) \/ (f1^f2)))

ChopOrImp: LEMMA  |-((f0^(f1 \/ f2)) => ((f0^f1) \/ (f0^f2)))

EmptyChop: LEMMA  |-((empty^f1) == f1)

ChopEmpty: LEMMA  |-((f1^empty) == f1)

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

StateImpBi: LEMMA  |-(p => Bi(p))

NextImpNotNextNot: LEMMA  |-(O(f0) => -O(-f0))

KeepnowImpNotKeepnowNot: LEMMA |-(keepn(f0) => -keepn(-f0))

BoxInduct: LEMMA |-((f0 /\ [](f0 => wO(f0))) => [](f0))

InfChop: LEMMA |-(((f0 /\ inf)^f1) == (f0 /\ inf))

ChopStarEqv: LEMMA
   |-(chopstar(f0) == (empty \/ ((f0 /\ more)^chopstar(f0))))

ChopstarInduct: LEMMA
   |-((inf /\ f0 /\ [](f0 => (f1 /\ fmore)^f0)) => chopstar(f1))

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

BoxGen: LEMMA |-(f0) IMPLIES |-([](f0))

BiGen : LEMMA |-(f0) IMPLIES |-(Bi(f0))
```

One has to prove the soundness of these axioms and rules before one can use them. The soundness of all the axioms and rules of ITL has been checked.

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):
      |-(f0 => f1) IMPLIES |-((f2 ^ f0) => (f2 ^ f1)))
```

- With skolimization we eliminate the for-all quantor.

```
Rule? (SKOSIMP)
Skolemizing and flattening, this simplifies to:
RightChopImpChop :
{-1}    |-(f0!1 => f1!1)
  |-------
{1}    |-((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}    |-([](f0!1 => f1!1))
[-2]    |-(f0!1 => f1!1)
  |-------
[1]    |-((f2!1 ^ f0!1) => (f2!1 ^ f1!1))
```

- Add an instance of lemma `BoxChopImpChop`. PVS will try to find the right instance.

```
Rule?
(USE "BoxChopImpChop")
Using lemma BoxChopImpChop, this simplifies to:
RightChopImpChop :
{-1}    |-([](f0!1 => f1!1) => (f2!1 ^ f0!1) => (f2!1 ^ f1!1))
[-2]    |-([](f0!1 => f1!1))
[-3]    |-(f0!1 => f1!1)
  |-------
[1]    |-((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  = 5.12 secs.
Real time = 12.10 secs.
```

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

3.3.2 First order ITL proof system encoding

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 data-types the following will do the job:

```
%%%% set of free state variables in an expresion
freeexp(e:exp) : RECURSIVE setof[(vr?)] =
    CASES e OF
     cst(n)                   : emptyset,
     vr(v,t)                  : singleton(vr(v,t)),
     ivr(iv,it,iexp)          : freeexp(iexp),
     one(oe,ot)               : freeexp(oe),
     two(te1,te2,tt)          : union(freeexp(te1),freeexp(te2)),
     four(fe1,fe2,fe3,fe4,ft) : union(union(freeexp(fe1), freeexp(fe2)),
                                      union(freeexp(fe3), freeexp(fe4)))
    ENDCASES
   MEASURE sizeexp(e)
%%%% set of free state variables in a formula
freeform(f:form) : RECURSIVE setof[(vr?)] =
    CASES f OF
     FA(va,fa)         : difference(freeform(fa), singleton(va)),
     skip              : emptyset,
     etwo(ee1,ee2,et)  : union(freeexp(ee1), freeexp(ee2)),
     fone(fo,for)      : freeform(fo),
     ftwo(ft1,ft2,ftt) : union(freeform(ft1), freeform(ft2))
     ENDCASES
   MEASURE sizeform(f)
```

Encoding of substitution is also relatively easy. In order to assure that expression *e* has the "same temporal data-type" 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 $\bigcirc$ and *fin* operators, and in case of static variables that *e* is also stable, i.e., doesn't change within an interval. This last condition can be directly encoded in ITL as a formula stable *exp*. That *e* doesn't contain $\bigcirc$ and *fin* can be encoded as follows:

```
%%%% definition of expressions containing no temporal constructs
  pexp         : TYPE =
     { e: exp | forall (oexp1: exp) :
                  forall (i: onetype) : not subterm(one(oexp1,i), e)}
```

Substitution is then encoded as follows:

```
%%%% definition of syntactic substitution in expression e1 of a variable
%%%% x by an expression pe containing no temporal constructs
su(e1, x, pe) : RECURSIVE exp =
    CASES e1 OF
     cst(n)                   : cst(n),
```

```
     vr(v,t)                  : if v(x)=v and t(x)=t
                                then pe else vr(v,t) endif,
     ivr(iv,it,iexp)          : ivr(iv,it,su(iexp, x, pe)),
     one(oe,ot)               : one(su(oe,x,pe),ot),
     two(te1,te2,tt)          : two(su(te1,x,pe),su(te2,x,pe),tt),
     four(fe1,fe2,fe3,fe4,ft) : four(su(fe1,x,pe),su(fe2,x,pe),
                                      su(fe3,x,pe),su(fe4,x,pe),ft)
     ENDCASES
   MEASURE sizeexp(e1)
%%%% definition of syntactic substitution in formula f1 of a  variable
%%%% x by an expression pe containing no temporal constructs
suform(f1, x, pe) : RECURSIVE form =
 CASES f1 OF
     FA(va,fa)        : FA(va,suform(fa,x,pe)),
     skip             : skip,
     etwo(ee1,ee2,et) : etwo(su(ee1,x,pe),su(ee2,x,pe),et),
     fone(fo,fot)     : fone(suform(fo,x,pe),fot),
     ftwo(ft1,ft2,ft) : ftwo(suform(ft1,x,pe),suform(ft2,x,pe),ft)
     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 following lemmas are crucial for proving the soundness of the first order axioms of ITL.

```
%%%% if a  variable doesn't occur in an expression then the value
%%%% of this expression doesn't depend on this  variable
  l_var_exp : LEMMA
   not member(v2,freeexp(e1)) implies
   if t(v2)=state then
    (forall val: infinite(val)=infinite(sigma) and len(val)=len(sigma)
           implies
    E(e1)(env,sigma)=
    E(e1)(env,(# infinite:=infinite(sigma),
                 len:=len(sigma),
                 seq:=(lambda (i:index[State](sigma)) :
                        ( PROJ_1(seq(sigma)(i)) with [(v(v2)):= seq(val)(i)],
                          PROJ_2(seq(sigma)(i))) ) #)))
   else
    (forall x1 :
        E(e1)(env,sigma) = E(e1)( (PROJ_1(env) with [(v(v2)):=x1],
                                   PROJ_2(env)), sigma))
   endif

%%%% syntactic subsitution of a  variable by an expression containing
%%%% no temporal constructs is the same as semantic substitution
```

```
l_sub_exp : LEMMA
    if t(x)=state then
      E(su(e1,x,pe))(env,sigma)=
      E(e1)(env,
            (# infinite:=infinite(sigma),
               len:=len(sigma),
               seq:=(lambda (i : index[State](sigma)):
                   (PROJ_1(seq(sigma)(i)) with
                     [(v(x)) := E(pe)(env,sub(sigma,i,i))],
                    PROJ_2(seq(sigma)(i))) ) #))
    else
      semstable(pe) implies
        E(su(e1,x,pe))(env,sigma) =
        E(e1)( (PROJ_1(env) with [(v(x)) := E(pe)(env,sigma)],
                PROJ_2(env)), sigma)
    endif

%%%% if a  variable doesn't occur in a formula then the truth
%%%% of this formula doesn't depend on this variable
  l_var_form : LEMMA
  not member(v2,freeform(f1))
  implies
  if t(v2)=state then
    (forall val: infinite(val)=infinite(sigma) and len(val)=len(sigma)
           implies
      M(f1)(env,sigma)=
      M(f1)(env,(# infinite:=infinite(sigma),
                   len:=len(sigma),
                   seq:=(lambda (i: index[State](sigma)):
                          (PROJ_1(seq(sigma)(i)) with
                          [(v(v2)):=seq(val)(i)],
                           PROJ_2(seq(sigma)(i))) ) #)))
  else
   (forall x1 :
        M(f1)(env,sigma) = M(f1)( (PROJ_1(env) with [(v(v2)):=x1],
                                   PROJ_2(env)), sigma))
  endif

%%%% syntactic substitution of a  variable by an expression containing
%%%% no temporal contructs is the same as semantic substitution provided
%%%% this  variable is not bound in f and this expression doesn't
%%%% contain bound variables of f
  l_sub_form : LEMMA
  not member(x, bound(f1)) and
  (forall (z:(vr?)):
    member(z,freeexp(pe)) implies not member(z,bound(f1)))
  implies
```

```
   if t(x)=state then
    M(suform(f1, x, pe))(env,sigma) =
    M(f1)(env,
      (# infinite:=infinite(sigma),
         len:=len(sigma),
         seq:=(lambda (i : index[State](sigma)):
             (PROJ_1(seq(sigma)(i)) with [(v(x))
                := E(pe)(env,sub(sigma,i,i))],
              PROJ_2(seq(sigma)(i))) #))
   else
    semstable(pe) implies
     M(suform(f1,x,pe))(env,sigma) =
     M(f1)( (PROJ_1(env) with [(v(x)) := E(pe)(env,sigma)],
             PROJ_2(env)), sigma)
   endif


%%%% a bound  variable of f can be renamed to a "fresh" bound variable
  l_ren : LEMMA
   not member(v1,bound(f1)) and
   not member(v2,bound(f1)) and
   v(v2) /= v(v1) and t(v1)=t(v2) and not member(v2,freeform(f1))
   implies
    M(FA(v1,f1))(env,sigma) = M(FA(v2,suform(f1,v1,v2)))(env,sigma)
```

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

```
%%%%%%%% first order axiom
ForallSub: LEMMA not member(v1,bound(f1)) and
     (t(v1)=static implies |-(stable(pe))) and
     (forall (z:(vr?)):
        member(z,freeexp(pe)) implies not member(z,bound(f1)))
     implies
          |-(FA(v1,f1) => suform(f1,v1,pe))

ForallImplies : LEMMA not member(v1,freeform(f1)) implies
          |-(FA(v1,f1=>f2) => (f1 => FA(v1,f2)))

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

ExistsChopLeft : LEMMA not member(v1,freeform(f1)) implies
          |-(TE(v1,f1^f2) => (f1^TE(v1,f2)))

%%%% the rule
ForallGen: LEMMA |-(f0) implies |-(FA(v1,f0))
```

Chapter 4

Refinement example

In this chapter we will present the refinement example. This example shows how ITL can be used for refinement using the work of [2]. The idea is that one starts with an abstract ITL specification and refine it into a concrete specification in this case Tempura code. We will use the PVS system to check the refinement steps.

4.1 Abstract specification

The specification that we want to refine is as follows:

$$
\begin{array}{l}
S_1 \stackrel{\text{def}}{=} \Box(\\
\qquad\qquad (x < 0 \wedge y = -1) \vee \\
\qquad\qquad (x = 0 \wedge y = 0) \vee \\
\qquad\qquad (x > 0 \wedge y = 1) \\
)
\end{array}
$$

The specification satisfies the following property:

$$P_1 \stackrel{\text{def}}{=} \Box(-1 \leq y \wedge y \leq 1).$$

I.e., the following must hold:

$$S_1 \quad \supset \quad P_1.$$

We will use PVS to prove this. First we have to transform above specification and property in "PVS/ITL"-language. First declaration of variables and constants:

```
x        :  (vr?) = vr(0,state)
y        :  (vr?) = vr(1,state)
minusone : (cst?) = cst(-1)
zero     : (cst?) = cst(0)
one      : (cst?) = cst(1)
```

The abstract specification and property then translated into:

```
%%%% sub-specification
 Choice : form = (x < zero /\ y = minusone) \/
                 (x = zero /\ y = zero    ) \/
                 (x > zero /\ y = one     )

%%%% first abstract (non-executable) specification
 S_1 : form = []( Choice )

%%%% property we want to prove
 P_1 : form = [](minusone <= y /\ y <= one)

%%%% first abstract (non-executable) specification satisfies the property
 l_prop   : LEMMA S_1 => P_1
```

We will first prove following lemma that will help us in proving `l_prop`.

```
%%%% help lemma for the proof of l_prop
 l_help_1 : LEMMA Choice => minusone <= y /\ y <= one
```

The proof of this lemma is as follows

```
l_help_1 :

  |-------
{1}    |-(Choice => minusone <= y /\ y <= one)
```

First tell PVS that it can rewrite all the definitions we have used for `S_1` and `P_1`.

```
Rerunning step: (AUTO-REWRITE-THEORY "demo")
Rewriting relative to the theory: demo,
this simplifies to:
l_help_1 :

  |-------
[1]    |-(Choice => minusone <= y /\ y <= one)
```

Give the semantic equivalent of above.

```
Rerunning step: (SEM)
giving the semantics,
this simplifies to:
l_help_1 :

  |-------
{1}    (FORALL (env: SState):
         (FORALL (sigma: Interval):
            PROJ_1(seq(sigma)(0))(0) < 0
             AND PROJ_1(seq(sigma)(0))(1) = -1
            OR PROJ_1(seq(sigma)(0))(0) = 0
               AND PROJ_1(seq(sigma)(0))(1) = 0
```

```
               OR 0 < PROJ_1(seq(sigma)(0))(0)
               AND PROJ_1(seq(sigma)(0))(1) = 1
           IMPLIES
             (-1 < PROJ_1(seq(sigma)(0))(1) OR
              -1 = PROJ_1(seq(sigma)(0))(1))
                AND
              (PROJ_1(seq(sigma)(0))(1) < 1 OR
               PROJ_1(seq(sigma)(0))(1) = 1)))
```

Eliminate the two for-all quantifiers.

```
Rerunning step: (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
l_help_1 :

{-1}    PROJ_1(seq(sigma!1)(0))(0) < 0 AND
        PROJ_1(seq(sigma!1)(0))(1) = -1
          OR PROJ_1(seq(sigma!1)(0))(0) = 0 AND
             PROJ_1(seq(sigma!1)(0))(1) = 0
            OR 0 < PROJ_1(seq(sigma!1)(0))(0) AND
               PROJ_1(seq(sigma!1)(0))(1) = 1
  |-------
{1}    (-1 < PROJ_1(seq(sigma!1)(0))(1) OR
        -1 = PROJ_1(seq(sigma!1)(0))(1))
          AND (PROJ_1(seq(sigma!1)(0))(1) < 1 OR
               PROJ_1(seq(sigma!1)(0))(1) = 1)
```

Use pre-defined PVS-function to prove this.

```
Rerunning step: (GROUND)
Applying propositional simplification and decision procedures,
Q.E.D.
```

The proof of `l_prop` proceeds then as follows:

```
l_prop :

  |-------
{1}    |-(S_1 => Prop)
```

Give the definition of `S_1`.

```
Rerunning step: (EXPAND "S_1")
Expanding the definition of S_1,
this simplifies to:
l_prop :

  |-------
{1}    |-(([](Choice)) => Prop))
```

```
}
Give the definition of \verb+P_1+.
{\small
\begin{verbatim}
Rerunning step: (EXPAND "P_1")
Expanding the definition of P_1,
this simplifies to:
l_prop :

  |-------
{1}    |-(([](Choice)) => [](minusone <= y /\ y <= one))))
```

Tell PVS that we can use the help lemma `l_help_1`.

```
Rerunning step: (LEMMA "l_help_1")
Applying l_help_1 where
this simplifies to:
l_prop :

{-1}    |-(Choice => minusone <= y /\ y <= one)
  |-------
[1]    |-(([](Choice)) => [](minusone <= y /\ y <= one))))
```

Use the `BoxImpBoxRule: |-(f0 => f1) implies |-([](f0) => [](f1))` of our list of valid rules/theorems to prove it. PVS will automatically try to find the correct instantiation of this rule.

```
Rerunning step: (REWRITE "BoxImpBoxRule")
Found matching substitution:
f1 gets (minusone <= y /\ y <= one),
f0 gets (Choice),
Rewriting using BoxImpBoxRule,
Q.E.D.
```

4.2 Refinement into concrete code

In this section we will refine `S_1` into concrete Tempura code. The first refinement step consists of adding interval length. The second refinement step consists of introduction of conditional and fixing the interval length to 5. The resulting specification is executable in Tempura, i.e., is it concrete. The third refinement step produces equal concrete code, it only introduces a “different” conditional.

4.2.1 The first refinement step

The first refinement step consists of adding interval length, i.e., `S_1` the interval length is arbitrary, in order to execute it one have to specify an interval length. In our case we specify that it should hold for interval length 5 or 6.

```
five     : (cst?) = cst(5)
six      : (cst?) = cst(6)

%%%% second more concrete specification
 S_2 : form = (intlen(five) \/ intlen(six)) /\ []( Choice )
```

Next we define the refinement operator $\sqsubseteq$ in PVS. Since it is a derived operator (namely from implication) it can be defined as follows:

```
f1,f2 : VAR form;
%%%% definition of refinement
 <=(f1,f2) : form = f2 => f1;
```

The following lemma expresses that S_2 is a refinement of S_1.

```
%%%% second non-exec. spec. is a refinement of first non-exec. spec.
 l_ref_1  : LEMMA S_1 <= S_2
```

The proof is as follows:

```
l_ref_1 :

  |-------
{1}    |-(S_1 <= S_2)
```

First we use the definition of S_2.

```
Rule? (EXPAND "S_2")
Expanding the definition of S_2,
this simplifies to:
l_ref_1 :

  |-------
{1}    |-((S_1 <= ((intlen(five) \/ intlen(six)) /\ [](Choice)))))
```

Then we use the definition of S_1.

```
Rule? (EXPAND "S_1")
Expanding the definition of S_1,
this simplifies to:
l_ref_1 :

  |-------
{1}    |-(([](Choice)) <= ((intlen(five) \/ intlen(six)) /\ [](Choice)))))
```

The we use the definition of refinement.

```
Rule? (EXPAND "<=")
Expanding the definition of <=,
this simplifies to:
l_ref_1 :

  |-------
{1}    |-(((intlen(five) \/ intlen(six)) /\ [](Choice)) => [](Choice))))
```

Use theorem Propeq12.

```
Rule? (REWRITE "Propeq12")
Found matching substitution:
f2 gets [](Choice)),
f1 gets [](Choice)),
f0 gets (intlen(five) \/ intlen(six)),
Rewriting using Propeq12,
this simplifies to:
l_ref_1 :

  |-------
{1}    |-((intlen(five) \/ intlen(six)) /\ [](Choice)) => (T /\ [](Choice))))
```

Use theorem Propimp17.

```
Rule? (REWRITE "Propimp17")
Found matching substitution:
f3 gets [](Choice)),
f1 gets T,
f2 gets [](Choice)),
f0 gets (intlen(five) \/ intlen(six)),
Rewriting using Propimp17,
this yields  2 subgoals:
l_ref_1.1 :

  |-------
{1}    |-((intlen(five) \/ intlen(six)) => T)
[2]    |-((intlen(five) \/ intlen(six)) /\ [](Choice)) => (T /\ [](Choice))))
```

Use theorem Prop8.

```
Rule? (REWRITE "Prop8")
Found matching substitution:
f0 gets (intlen(five) \/ intlen(six)),
Rewriting using Prop8,

This completes the proof of l_ref_1.1.

l_ref_1.2 :

  |-------
{1}    |-([](Choice)) => [](Choice)))
[2]    |-((intlen(five) \/ intlen(six)) /\ [](Choice)) => (T /\ [](Choice))))
```

Use theorem Prop10.

```
Rule? (REWRITE "Prop10")
Found matching substitution:
f0 gets [](Choice)),
```

```
Rewriting using Prop10,

This completes the proof of l_ref_1.2.

Q.E.D.
```

4.2.2 The second refinement step

The second refinement step consists of introduction of the conditional and fixing the interval length to 5. This specification is executable in Tempura.

```
%%%%  sub-spec. of first executable specification
 If1 : form =  ife(x < zero, y = minusone,
                   ife(x = zero, y = zero, y = one))

%%%% first executable specification
 S_exec_1 : form = intlen(five) /\ []( If1 )
```

The proof of the second refinement step uses following refinement law.

```
%%%% refinement law used in l_ref_2
 l_help_2 : LEMMA    ((f3 => -f1) /\ ((-f1 /\ -f3) => f6)) =>
        (((f1 /\ f2) \/ (f3 /\ f4) \/ (f6 /\ f5)) <=
         ife(f1,f2,ife(f3,f4,f5)) )
```

The proof of this refinement law is as follows:

```
l_help_2 :

  |-------
{1}    (FORALL (f1: form, f2: form, f3: form, f4: form, f5: form, f6: form):
         |-(((f3 => -f1) /\ ((-f1 /\ -f3) => f6))
                =>
             (((f1 /\ f2) \/ (f3 /\ f4) \/ (f6 /\ f5))
                  <= ife(f1, f2, ife(f3, f4, f5)))))
```

Use the definition of refinement.

```
Rule? (EXPAND "<=")
Expanding the definition of <=,
this simplifies to:
l_help_2 :

  |-------
{1}    (FORALL (f1: form, f2: form, f3: form, f4: form, f5: form, f6: form):
         |-((((f3 => -f1) /\ ((-f1 /\ -f3) => f6))
                =>
              (ife(f1, f2, ife(f3, f4, f5))
                   => ((f1 /\ f2) \/ (f3 /\ f4) \/ (f6 /\ f5))))))
```

Give the semantics of resulting formula.

```
Rule? (SEM)
giving the semantics,
this simplifies to:
l_help_2 :

  |-------
{1}    (FORALL (f1: form, f2: form, f3: form, f4: form, f5: form, f6: form):
         (FORALL (env: SState):
            (FORALL (sigma: Interval):
               ((M(f3)(env, sigma) IMPLIES NOT M(f1)(env, sigma))
                    AND
                  (NOT M(f1)(env, sigma) AND NOT M(f3)(env, sigma)
                       IMPLIES M(f6)(env, sigma)))
                IMPLIES M(f1)(env, sigma) AND M(f2)(env, sigma)
                  OR NOT M(f1)(env, sigma)
                    AND
                  (M(f3)(env, sigma) AND M(f4)(env, sigma)
                       OR NOT M(f3)(env, sigma) AND M(f5)(env, sigma))
                  IMPLIES M(f1)(env, sigma) AND M(f2)(env, sigma)
                    OR M(f3)(env, sigma) AND M(f4)(env, sigma)
                      OR M(f6)(env, sigma) AND M(f5)(env, sigma))))
```

Eliminate the forall-quantifiers and simplify.

```
Rule? (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
l_help_2 :

{-1}    M(f3!1)(env!1, sigma!1) IMPLIES NOT M(f1!1)(env!1, sigma!1)
{-2}    NOT M(f1!1)(env!1, sigma!1) AND NOT M(f3!1)(env!1, sigma!1)
          IMPLIES M(f6!1)(env!1, sigma!1)
{-3}    M(f1!1)(env!1, sigma!1) AND M(f2!1)(env!1, sigma!1)
          OR NOT M(f1!1)(env!1, sigma!1)
            AND
          (M(f3!1)(env!1, sigma!1) AND M(f4!1)(env!1, sigma!1)
               OR NOT M(f3!1)(env!1, sigma!1) AND M(f5!1)(env!1, sigma!1))
  |-------
{1}    M(f1!1)(env!1, sigma!1) AND M(f2!1)(env!1, sigma!1)
{2}    M(f3!1)(env!1, sigma!1) AND M(f4!1)(env!1, sigma!1)
{3}    M(f6!1)(env!1, sigma!1) AND M(f5!1)(env!1, sigma!1)
```

Use propositional simplification.

```
Rule? (PROP)
Applying propositional simplification,
Q.E.D.
```

The proof of refinement, i.e., the following lemma

```
%%%% first exec. spec. is a refinement of second non-exec. spec.
 l_ref_2  : LEMMA S_2 <= S_exec_1
```

is as follows:

```
l_ref_2 :

  |-------
{1}    |-(S_2 <= S_exec_1)
```

Use the definition of S_exec_1.

```
Rule? (EXPAND "S_exec_1")
Expanding the definition of S_exec_1,
this simplifies to:
l_ref_2 :

  |-------
{1}    |-((S_2 <= (intlen(five) /\ [](If1)))))
```

Use the definition of S_2.

```
Rule? (EXPAND "S_2")
Expanding the definition of S_2,
this simplifies to:
l_ref_2 :

  |-------
{1}    |-((((intlen(five) \/ intlen(six)) /\ [](Choice)))
              <= (intlen(five) /\ [](If1)))))
```

Use the definition of refinement.

```
Rule? (EXPAND "<=")
Expanding the definition of <=,
this simplifies to:
l_ref_2 :

  |-------
{1}    |-((intlen(five) /\ [](If1))
              => (intlen(five) \/ intlen(six)) /\ [](Choice))))
```

Use theorem Propimp17.

```
Rule? (REWRITE "Propimp17")
Found matching substitution:
f3 gets [](Choice)),
f1 gets (intlen(five) \/ intlen(six)),
f2 gets [](If1)),
```

```
f0 gets intlen(five),
Rewriting using Propimp17,
this yields  2 subgoals:
l_ref_2.1 :

  |-------
{1}    |-(intlen(five) => (intlen(five) \/ intlen(six)))
[2]    |-((intlen(five) /\ [](If1))
              => (intlen(five) \/ intlen(six)) /\ [](Choice))))
```

Use theorem `Prop11`.

```
Rule? (REWRITE "Prop11")
Found matching substitution:
f1 gets intlen(six),
f0 gets intlen(five),
Rewriting using Prop11,

This completes the proof of l_ref_2.1.

l_ref_2.2 :

  |-------
{1}    |-([](If1)) => [](Choice)))
[2]    |-((intlen(five) /\ [](If1))
              => (intlen(five) \/ intlen(six)) /\ [](Choice))))
```

Hide formula 2 because we want to prove formula 1.

```
Rule? (HIDE 2)
Hiding formulas:  2,
this simplifies to:
l_ref_2.2 :

  |-------
[1]    |-([](If1)) => [](Choice)))
```

Use the definition of `If1`.

```
Rule? (EXPAND "If1")
Expanding the definition of If1,
this simplifies to:
l_ref_2.2 :

  |-------
{1}    |-(([](ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one)))
              => [](Choice))))
```

Use the definition of `Choice`.

```
Rule? (EXPAND "Choice")
Expanding the definition of Choice,
this simplifies to:
l_ref_2.2 :

  |-------
{1}    |-((([](ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one)))
              =>
            [](((x < zero /\ y = minusone)
                   \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one))))
```

Introduce the above refinement law.

```
Rule? (LEMMA "l_help_2")
Applying l_help_2 where
this simplifies to:
l_ref_2.2 :

{-1}    (FORALL (f1: form, f2: form, f3: form, f4: form, f5: form, f6: form):
         |-(((f3 => -f1) /\ ((-f1 /\ -f3) => f6))
                =>
              (((f1 /\ f2) \/ (f3 /\ f4) \/ (f6 /\ f5))
                   <= ife(f1, f2, ife(f3, f4, f5)))))
  |-------
[1]    |-((([](ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one)))
              =>
            [](((x < zero /\ y = minusone)
                   \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one))))
```

Use the definition of refinement.

```
Rule? (EXPAND "<=")
Expanding the definition of <=,
this simplifies to:
l_ref_2.2 :

{-1}    (FORALL (f1: form, f2: form, f3: form, f4: form, f5: form, f6: form):
         |-((((f3 => -f1) /\ ((-f1 /\ -f3) => f6))
                 =>
               (ife(f1, f2, ife(f3, f4, f5))
                    => ((f1 /\ f2) \/ (f3 /\ f4) \/ (f6 /\ f5))))))
  |-------
[1]    |-((([](ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one)))
              =>
            [](((x < zero /\ y = minusone)
                   \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one))))
```

Instantiate the refinement law.

```
Rule? (INST?)
Found substitution:
f5 gets y = one,
f6 gets x > zero,
f4 gets y = zero,
f3 gets x = zero,
f2 gets y = minusone,
f1 gets x < zero,
Instantiating quantified variables,
this simplifies to:
l_ref_2.2 :

{-1}    |-((((x = zero => -(x < zero))
               /\ ((-(x < zero) /\ -(x = zero)) => x > zero))
              =>
             (ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
                  =>
                ((x < zero /\ y = minusone)
                     \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one)))))
  |-------
[1]    |-(([](ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one)))
              =>
             [](((x < zero /\ y = minusone)
                    \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one))))
```

Use theorem `BoxImpBoxRule`.

```
Rule? (REWRITE "BoxImpBoxRule")
Found matching substitution:
f1 gets (x < zero /\ y = minusone) \/ (x = zero /\ y = zero) \/
        (x > zero /\ y = one),
f0 gets ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one)),
Rewriting using BoxImpBoxRule,
this simplifies to:
l_ref_2.2 :

[-1]    |-((((x = zero => -(x < zero))
               /\ ((-(x < zero) /\ -(x = zero)) => x > zero))
              =>
             (ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
                  =>
                ((x < zero /\ y = minusone)
                     \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one)))))
  |-------
{1}    |-(ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
             => (x < zero /\ y = minusone)
               \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one))
[2]    |-(([](ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one)))
```

```
              =>
            [](((x < zero /\ y = minusone)
                   \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one))))
```

Hide formula 2 because we want to prove formula 1.

```
Rule? (HIDE 2)
Hiding formulas:  2,
this simplifies to:
l_ref_2.2 :

[-1]    |-((((x = zero => -(x < zero))
               /\ ((-(x < zero) /\ -(x = zero)) => x > zero))
              =>
             (ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
                  =>
               ((x < zero /\ y = minusone)
                    \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one)))))
  |-------
[1]    |-(ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
             => (x < zero /\ y = minusone)
               \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one))
```

Use the Modus Ponus rule.

```
Rule? (USE "MP"
        ("f0"
         "((x = zero => -(x < zero))
             /\\ ((-(x < zero) /\\ -(x = zero)) => x > zero))"))
Using lemma MP,
this simplifies to:
l_ref_2.2 :

{-1}    |-(((x = zero => -((x < zero)))
              /\ ((-((x < zero)) /\ -((x = zero))) => x > zero))
             => ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
               => (x < zero /\ y = minusone)
                 \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one))
          AND
        |-(((x = zero => -((x < zero)))
               /\ ((-((x < zero)) /\ -((x = zero))) => x > zero)))
          IMPLIES
        |-(ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
              => (x < zero /\ y = minusone)
                \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one))
[-2]    |-((((x = zero => -(x < zero))
              /\ ((-(x < zero) /\ -(x = zero)) => x > zero))
             =>
            (ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
```

```
                =>
              ((x < zero /\ y = minusone)
                   \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one)))))
  |-------
[1]    |-(ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
             => (x < zero /\ y = minusone)
               \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one))
```

Simplify the resulting formulae.

```
Rule? (ASSERT)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
l_ref_2.2 :

[-1]    |-((((x = zero => -(x < zero))
               /\ ((-(x < zero) /\ -(x = zero)) => x > zero))
              =>
            (ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
                 =>
               ((x < zero /\ y = minusone)
                    \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one)))))
  |-------
{1}    |-(((x = zero => -((x < zero)))
             /\ ((-((x < zero)) /\ -((x = zero))) => x > zero)))
[2]    |-(ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
             => (x < zero /\ y = minusone)
               \/ (x = zero /\ y = zero) \/ (x > zero /\ y = one))
```

Hide formula -1 and 2 because we want to prove formula 1.

```
Rule? (HIDE -1 2)
Hiding formulas:  -1, 2,
this simplifies to:
l_ref_2.2 :

  |-------
[1]    |-(((x = zero => -((x < zero)))
             /\ ((-((x < zero)) /\ -((x = zero))) => x > zero)))
```

Tell PVS to automatically rewrite the definitions introduced in theory demo.

```
Rule? (AUTO-REWRITE-THEORY "demo")
Rewriting relative to the theory: demo,
this simplifies to:
l_ref_2.2 :

  |-------
[1]    |-(((x = zero => -((x < zero)))
             /\ ((-((x < zero)) /\ -((x = zero))) => x > zero)))
```

Give the semantics of above formula.

```
Rule? (SEM)
giving the semantics,
this simplifies to:
l_ref_2.2 :

  |-------
{1}    (FORALL (env: SState):
         (FORALL (sigma: Interval):
            (PROJ_1(seq(sigma)(0))(0) = 0
                 IMPLIES NOT PROJ_1(seq(sigma)(0))(0) < 0)
                AND
              (NOT PROJ_1(seq(sigma)(0))(0) < 0
                   AND NOT PROJ_1(seq(sigma)(0))(0) = 0
                   IMPLIES 0 < PROJ_1(seq(sigma)(0))(0))))
```

Eliminate the forall-quantifiers and simplify.

```
Rule? (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
l_ref_2.2 :

  |-------
{1}    (PROJ_1(seq(sigma!1)(0))(0) = 0
           IMPLIES NOT PROJ_1(seq(sigma!1)(0))(0) < 0)
          AND
        (NOT PROJ_1(seq(sigma!1)(0))(0) < 0
             AND NOT PROJ_1(seq(sigma!1)(0))(0) = 0
             IMPLIES 0 < PROJ_1(seq(sigma!1)(0))(0))
```

Use propositional simplification and use arithmetic.

```
Rule? (GROUND)
Applying propositional simplification and decision procedures,

This completes the proof of l_ref_2.2.

Q.E.D.
```

4.2.3 The third refinement step

The third refinement step produces equal concrete code, i.e., it introduces a different conditional. The specification is as follows.

```
%%%% sub-spec. of second executable specification
 If2 : form =  ife(x = zero, y = zero,
                   ife(x > zero, y = one, y = minusone))
```

```
%%%% second executable specification
 S_exec_2 : form = intlen(five) /\ []( If2 )
```

The proof that these concrete code specifications are equal uses the following algebraic law.

```
%%%% algebraic law used in l_equal
 l_help_3 : LEMMA (f1 => -f3) /\ (f6 == (-f1 /\ -f3)) =>
           (ife(f1,f2,ife(f3,f4,f5)) == ife(f3,f4,ife(f6,f5,f2)))
```

The proof of this law is as follows:

```
l_help_3 :

  |-------
{1}    (FORALL (f1: form, f2: form, f3: form, f4: form, f5: form, f6: form):
         |-((f1 => -f3) /\ (f6 == (-f1 /\ -f3))
                =>
              (ife(f1, f2, ife(f3, f4, f5)) == ife(f3, f4, ife(f6, f5, f2)))))
```

Give the semantics of this formula.

```
Rule? (SEM)
giving the semantics,
this simplifies to:
l_help_3 :

  |-------
{1}    (FORALL (f1: form, f2: form, f3: form, f4: form, f5: form, f6: form):
         (FORALL (env: SState):
            (FORALL (sigma: Interval):
               ((M(f1)(env, sigma) IMPLIES NOT M(f3)(env, sigma))
                    AND
                  (M(f6)(env, sigma)
                       IMPLIES NOT M(f1)(env, sigma) AND NOT M(f3)(env, sigma))
                    AND
                  (NOT M(f1)(env, sigma) AND NOT M(f3)(env, sigma)
                       IMPLIES M(f6)(env, sigma)))
                 IMPLIES
                (M(f1)(env, sigma) AND M(f2)(env, sigma)
                    OR NOT M(f1)(env, sigma)
                      AND
                    (M(f3)(env, sigma) AND M(f4)(env, sigma)
                         OR NOT M(f3)(env, sigma) AND M(f5)(env, sigma))
                    IMPLIES M(f3)(env, sigma) AND M(f4)(env, sigma)
                      OR NOT M(f3)(env, sigma)
                        AND
                      (M(f6)(env, sigma) AND M(f5)(env, sigma)
                           OR NOT M(f6)(env, sigma) AND M(f2)(env, sigma)))
                   AND
```

```
                    (M(f3)(env, sigma) AND M(f4)(env, sigma)
                         OR NOT M(f3)(env, sigma)
                           AND
                         (M(f6)(env, sigma) AND M(f5)(env, sigma)
                              OR NOT M(f6)(env, sigma) AND M(f2)(env, sigma))
                         IMPLIES M(f1)(env, sigma) AND M(f2)(env, sigma)
                           OR NOT M(f1)(env, sigma)
                             AND
                           (M(f3)(env, sigma) AND M(f4)(env, sigma)
                                OR NOT M(f3)(env, sigma)
                                  AND M(f5)(env, sigma))))))
```

Eliminate the forall-quantifiers and simplify.

```
Rule? (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
l_help_3 :

{-1}    M(f1!1)(env!1, sigma!1) IMPLIES NOT M(f3!1)(env!1, sigma!1)
{-2}    M(f6!1)(env!1, sigma!1)
          IMPLIES NOT M(f1!1)(env!1, sigma!1) AND NOT M(f3!1)(env!1, sigma!1)
{-3}    NOT M(f1!1)(env!1, sigma!1) AND NOT M(f3!1)(env!1, sigma!1)
          IMPLIES M(f6!1)(env!1, sigma!1)
  |-------
{1}    (M(f1!1)(env!1, sigma!1) AND M(f2!1)(env!1, sigma!1)
           OR NOT M(f1!1)(env!1, sigma!1)
             AND
           (M(f3!1)(env!1, sigma!1) AND M(f4!1)(env!1, sigma!1)
                OR NOT M(f3!1)(env!1, sigma!1) AND M(f5!1)(env!1, sigma!1))
           IMPLIES M(f3!1)(env!1, sigma!1) AND M(f4!1)(env!1, sigma!1)
             OR NOT M(f3!1)(env!1, sigma!1)
               AND
             (M(f6!1)(env!1, sigma!1) AND M(f5!1)(env!1, sigma!1)
                  OR NOT M(f6!1)(env!1, sigma!1) AND M(f2!1)(env!1, sigma!1)))
         AND
        (M(f3!1)(env!1, sigma!1) AND M(f4!1)(env!1, sigma!1)
             OR NOT M(f3!1)(env!1, sigma!1)
               AND
             (M(f6!1)(env!1, sigma!1) AND M(f5!1)(env!1, sigma!1)
                  OR NOT M(f6!1)(env!1, sigma!1) AND M(f2!1)(env!1, sigma!1))
             IMPLIES M(f1!1)(env!1, sigma!1) AND M(f2!1)(env!1, sigma!1)
               OR NOT M(f1!1)(env!1, sigma!1)
                 AND
               (M(f3!1)(env!1, sigma!1) AND M(f4!1)(env!1, sigma!1)
                    OR NOT M(f3!1)(env!1, sigma!1) AND M(f5!1)(env!1, sigma!1)))
```

Do propositional simplification.

```
Rule? (PROP)
Applying propositional simplification,
Q.E.D.
```

The proof that the two concrete code specifications are equal, i.e., the following

```
%%%% first and second exec. spec. are equal
 l_equal  : LEMMA S_exec_1 == S_exec_2
```

is as follows:

```
l_equal :

  |-------
{1}    |-(S_exec_1 == S_exec_2)
```

Use the definition of S_exec_1.

```
Rule? (EXPAND "S_exec_1")
Expanding the definition of S_exec_1,
this simplifies to:
l_equal :

  |-------
{1}    |-(((intlen(five) /\ [](If1))) == S_exec_2))
```

Use the definition of S_exec_2.

```
Rule? (EXPAND "S_exec_2")
Expanding the definition of S_exec_2,
this simplifies to:
l_equal :

  |-------
{1}    |-(((intlen(five) /\ [](If1))) == (intlen(five) /\ [](If2)))))
```

Use theorem Propimp16.

```
Rule? (REWRITE "Propimp16")
Found matching substitution:
f3 gets [](If2)),
f1 gets intlen(five),
f2 gets [](If1)),
f0 gets intlen(five),
Rewriting using Propimp16,
this yields  2 subgoals:
l_equal.1 :

  |-------
{1}    |-(intlen(five) == intlen(five))
[2]    |-(((intlen(five) /\ [](If1))) == (intlen(five) /\ [](If2)))))
```

Use theorem Prop7.

```
Rule? (REWRITE "Prop7")
Found matching substitution:
f0 gets intlen(five),
Rewriting using Prop7,

This completes the proof of l_equal.1.

l_equal.2 :

  |-------
{1}    |-([]((If1)) == []((If2)))
[2]    |-(((intlen(five) /\ []((If1))) == (intlen(five) /\ []((If2)))))
```

Hide formula 2 because we want to prove formula 1.

```
Rule? (HIDE 2)
Hiding formulas:  2,
this simplifies to:
l_equal.2 :

  |-------
[1]    |-([]((If1)) == []((If2)))
```

Use the definition of If1.

```
Rule? (EXPAND "If1")
Expanding the definition of If1,
this simplifies to:
l_equal.2 :

  |-------
{1}    |-(([](ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one)))
              == []((If2))))
```

Use the definition of If2.

```
Rule? (EXPAND "If2")
Expanding the definition of If2,
this simplifies to:
l_equal.2 :

  |-------
{1}    |-(([](ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one)))
              ==
            [](ife(x = zero, y = zero,
                   ife(x > zero, y = one, y = minusone)))))
```

Use the algebraic law of above.

```
Rule? (USE "l_help_3")
Using lemma l_help_3,
this simplifies to:
l_equal.2 :

{-1}    |-((x < zero => -(x = zero))
             /\ (x > zero == (-(x < zero) /\ -(x = zero)))
             =>
           (ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
                ==
              ife(x = zero, y = zero, ife(x > zero, y = one, y = minusone))))
  |-------
[1]    |-(([](ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one)))
              ==
            [](ife(x = zero, y = zero,
                   ife(x > zero, y = one, y = minusone)))))
```

Use theorem BoxEqvBox.

```
Rule? (REWRITE "BoxEqvBox")
Found matching substitution:
f1 gets ife(x = zero, y = zero, ife(x > zero, y = one, y = minusone)),
f0 gets ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one)),
Rewriting using BoxEqvBox,
this simplifies to:
l_equal.2 :

[-1]    |-((x < zero => -(x = zero))
             /\ (x > zero == (-(x < zero) /\ -(x = zero)))
             =>
           (ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
                ==
              ife(x = zero, y = zero, ife(x > zero, y = one, y = minusone))))
  |-------
{1}    |-(ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
             == ife(x = zero, y = zero, ife(x > zero, y = one, y = minusone)))
[2]    |-(([](ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one)))
              ==
            [](ife(x = zero, y = zero,
                   ife(x > zero, y = one, y = minusone)))))
```

Hide formula 2 because we want to prove formula 1.

```
Rule? (HIDE 2)
Hiding formulas:  2,
this simplifies to:
l_equal.2 :

[-1]    |-((x < zero => -(x = zero))
```

```
             /\ (x > zero == (-(x < zero) /\ -(x = zero)))
             =>
           (ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
                ==
            ife(x = zero, y = zero, ife(x > zero, y = one, y = minusone))))
  |-------
[1]    |-(ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
             == ife(x = zero, y = zero, ife(x > zero, y = one, y = minusone)))
```

Use the Modus Ponus.

```
Rule? (USE "MP")
Using lemma MP,
this simplifies to:
l_equal.2 :

{-1}    |-((x < zero => -(x = zero))
             /\ (x > zero == (-(x < zero) /\ -(x = zero)))
             =>
           (ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
                ==
            ife(x = zero, y = zero, ife(x > zero, y = one, y = minusone))))
          AND
        |-((x < zero => -(x = zero))
              /\ (x > zero == (-(x < zero) /\ -(x = zero))))
          IMPLIES
        |-((ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
                ==
             ife(x = zero, y = zero,
                 ife(x > zero, y = one, y = minusone))))
[-2]    |-((x < zero => -(x = zero))
             /\ (x > zero == (-(x < zero) /\ -(x = zero)))
             =>
           (ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
                ==
            ife(x = zero, y = zero, ife(x > zero, y = one, y = minusone))))
  |-------
[1]    |-(ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
             == ife(x = zero, y = zero, ife(x > zero, y = one, y = minusone)))
```

Simplify the resulting formulae.

```
Rule? (ASSERT)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
l_equal.2 :

[-1]    |-((x < zero => -(x = zero))
             /\ (x > zero == (-(x < zero) /\ -(x = zero)))
```

```
            =>
          (ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
              ==
           ife(x = zero, y = zero, ife(x > zero, y = one, y = minusone))))
  |-------
{1}    |-((x < zero => -(x = zero))
            /\ (x > zero == (-(x < zero) /\ -(x = zero))))
[2]    |-(ife(x < zero, y = minusone, ife(x = zero, y = zero, y = one))
            == ife(x = zero, y = zero, ife(x > zero, y = one, y = minusone)))
```

Hide formula -1 and 2 because we want to prove formula 1.

```
Rule? (HIDE -1 2)
Hiding formulas:  -1, 2,
this simplifies to:
l_equal.2 :

  |-------
[1]    |-((x < zero => -(x = zero))
            /\ (x > zero == (-(x < zero) /\ -(x = zero))))
```

Tell PVS to automatically use the definitions of theory demo.

```
Rule? (AUTO-REWRITE-THEORY "demo")
Rewriting relative to the theory: demo,
this simplifies to:
l_equal.2 :

  |-------
[1]    |-((x < zero => -(x = zero))
            /\ (x > zero == (-(x < zero) /\ -(x = zero))))
```

Give the semantics of this formula.

```
Rule? (SEM)
giving the semantics,
this simplifies to:
l_equal.2 :

  |-------
{1}    (FORALL (env: SState):
         (FORALL (sigma: Interval):
            (PROJ_1(seq(sigma)(0))(0) < 0
                 IMPLIES NOT PROJ_1(seq(sigma)(0))(0) = 0)
               AND
              (0 < PROJ_1(seq(sigma)(0))(0)
                   IMPLIES NOT PROJ_1(seq(sigma)(0))(0) < 0
                     AND NOT PROJ_1(seq(sigma)(0))(0) = 0)
                 AND
               (NOT PROJ_1(seq(sigma)(0))(0) < 0
```

```
                    AND NOT PROJ_1(seq(sigma)(0))(0) = 0
                    IMPLIES 0 < PROJ_1(seq(sigma)(0))(0))))
```

Eliminate the forall-quantifiers and simplify.

```
Rule? (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
l_equal.2 :

  |-------
{1}    (PROJ_1(seq(sigma!1)(0))(0) < 0
           IMPLIES NOT PROJ_1(seq(sigma!1)(0))(0) = 0)
          AND
        (0 < PROJ_1(seq(sigma!1)(0))(0)
             IMPLIES NOT PROJ_1(seq(sigma!1)(0))(0) < 0
               AND NOT PROJ_1(seq(sigma!1)(0))(0) = 0)
            AND
          (NOT PROJ_1(seq(sigma!1)(0))(0) < 0
               AND NOT PROJ_1(seq(sigma!1)(0))(0) = 0
               IMPLIES 0 < PROJ_1(seq(sigma!1)(0))(0))
```

Use propositional simplification and arithmetic.

```
Rule? (GROUND)
Applying propositional simplification and decision procedures,

This completes the proof of l_equal.2.

Q.E.D.
```

Chapter 5

Conclusion and future work

We have used the ITL proof assistant to verify a list of more than 100 theorems. Experience shows that proofs within the tool in general follow the pattern as the "by hand" case. This ensures that people who are used to the proofs "by hand" can easily make the switch to 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). The syntax of ITL encoded so far in PVS enables us to write this specification in PVS. We will prove the correctness of this example by using refinement, i.e., first give an abstract specification and prove that is satisfies certain properties. Then use refinement rules to show that the big (concrete) specification is a refinement of this abstract specification. In [3] we have already proved that an abstract specification of the EP/3 satisfies certain properties. This proof however is done by hand, so this proof is checked first. This will 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 need to be encoded because the proof uses these compositional rules. These proof rules are discussed in [6, 7]. Because the basic ITL formalism is now encoded the encoding of these compositional proof rules is straight forward. We will use the refinement rules defined in [2] to prove refinement. The encoding of these refinement rules in PVS is also straight forward.

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[4]. 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 data-types 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.

Bibliography

[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. Refining Interval Temporal Logic Specifications. In proc. of Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software (ARTS'97), Mallorca, Spain, May 21-23, 1997.

[3] X. Li, A. Cau, B. Moszkowski, N. Coleman and H. Zedan. Proving the Correctness of the Interlock Mechanism in Processor Design. Submitted to FME'97.

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

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

[6] 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.

[7] 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.

[8] 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`

[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.

[10] 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.

Appendix A

Translation of ITL into PVS

ITL	PVS
$\neg f_1$	`-f1`
f_1^*	`chopstar(f1)`
$f_1 \wedge f_2$	`f1 /\ f2`
$f_1 \vee f_2$	`f1 \/ f2`
$f_1 \supset f_2$	`f1 => f2`
$f_1 ; f_2$	`f1^f2`
$\exists va \cdot f_1$	`TE(va,f1)`
$\forall va \cdot f_1$	`FA(va,f1)`
true	`T`
false	`F`
$\bigcirc f_1$	`O(f1)`
$f_1 \equiv f_2$	`f1 == f2`
$e_1 + e_2$	`e1+e2`
$e_1 - e_2$	`e1-e2`
$e_1 * e_2$	`e1*e2`
$e_1 \div e_2$	`div(e1,e2)`
$e_1 \, mod \, e_2$	`mod(e1,e2)`
$e_1 = e_2$	`e1 = e2`
$e_1 < e_2$	`e1 < e2`
$e_1 \leq e_2$	`e1 <= e2`
$e_1 > e_2$	`e1 > e2`
$e_1 \geq e_2$	`e1 >= e2`
$e_1 \neq e_2$	`e1 /= e2`
more	`more`
empty	`empty`
inf	`inf`
finite	`finite`
fmore	`fmore`
$isinf(f_1)$	`isinf(f1)`
$isfin(f_1)$	`isfin(f1)`
$\Diamond f_1$	`<>(f1)`

ITL	PVS
$\Box f_1$	`[](f1)`
$\textcircled{w} f_1$	`wO(f1)`
$\textcircled{i}(f_1)$	`Di(f1)`
$\boxed{i}(f_1)$	`Bi(f1)`
$\textcircled{a}(f_1)$	`Da(f1)`
$\boxed{a}(f_1)$	`Ba(f1)`
$\textcircled{m}(f_1)$	`Dm(f1)`
$\boxed{m}(f_1)$	`Bm(f1)`
if f_0 then f_1 else f_2	`ife(f0, f1, f2)`
if f_0 then f_1	`ifo(f0, f1)`
$fin(f_1)$	`fin(f1)`
$sfin(f_1)$	`sfin(f1)`
$halt(f_1)$	`halt(f1)`
$shalt(f_1)$	`shalt(f1)`
$keep(f_1)$	`keep(f1)`
$keepnow(f_1)$	`keepn(f1)`
$\bigcirc e_1$	`nx(e1)`
fin e_1	`fin(e1)`
$e_1 := e_2$	`as(e1,e2)`
$e_1 \approx e_2$	`equal(e1,e2)`
$e_1 \leftarrow e_2$	`tas(e1, e2)`
e_1 *gets* e_2	`gets(e1, e2)`
stable e_1	`stable(e1)`
padded e_1	`padded(e1)`
$e_1 <\!\sim e_2$	`pta(e1, e2)`
goodindex e_1	`goodindex(e1)`
$intlen(e_1)$	`intlen(e1)`
f_1^ω	`omega(f1)`
$fstar(f_1)$	`fstar(f1)`
while f_0 do f_1	`while(f0, f1)`
repeat f_0 until f_1	`repeat(f0, f1)`

Appendix B

ITL Rules and Theorems in PVS

```
%%%% the axioms
ChopAssoc:                      ((f0^f1)^f2) == (f0^(f1^f2))
OrChopImp:                      ((f0 \/ f1)^f2) => ((f0^f2) \/ (f1^f2))
ChopOrImp:                      (f0^(f1 \/ f2)) => ((f0^f1) \/ (f0^f2))
EmptyChop:                      (empty^f1) == f1
ChopEmpty:                      (f1^empty) == f1
BiBoxChopImpChop:               (Bi(f0 => f1) /\ [](f2 => f3)) => ((f0^f2) => (f1^f3))
StateImpBi:                     p => Bi(p)
NextImpNotNextNot:              O(f0) => -O(-f0)
KeepnowImpNotKeepnowNot:        keepn(f0) => -keepn(-f0)
BoxInduct:                      (f0 /\ [](f0 => wO(f0))) => [](f0)
InfChop:                        ((f0 /\ inf)^f1) == (f0 /\ inf)
ChopStarEqv:                    chopstar(f0) == (empty \/ ((f0 /\ more)^chopstar(f0)))
ChopstarInduct:                 (inf /\ f0 /\ [](f0 => (f1 /\ fmore)^f0)) => chopstar(f1)

%%%% the rules
MP:                             |-(f0 => f1) AND |-(f0) IMPLIES |-(f1)
BoxGen:                         |-(f0) IMPLIES |-([](f0))
BiGen:                          |-(f0) IMPLIES |-(Bi(f0))

%%%% basic axioms/rewrite rules
SkipFinite:                     skip => finite
FiniteChopTrue:                 finite^T
InfEqvSkipChopInf:              inf == (skip^inf)
FiniteChopFiniteEqvFinite:      (finite^finite) == finite
FiniteAndMoreEqvFiniteChopSkip: (finite /\ more) == (finite^skip)
FiniteAndMoreEqvSkipChopFinite: (finite /\ more) == (skip^finite)
SkipChopFiniteImpFinite:        (skip^finite) => finite
FiniteChopSkipImpFinite:        (finite^skip) => finite
FiniteChopInfEqvInfinite:       (finite^inf) == inf
EmptyFinite:                    empty => finite
InfChopFiniteEqvInf:            (inf^finite) == inf
InfChopInfEqvInf:               (inf^inf) == inf

%%%% some usefull propositional lemma's and rewrite rules
Prop1     :  (f0 \/ f0) => f0
Prop2     :  f1 => (f0 \/ f1)
Prop3     :  (f0 \/ f1) => (f1 \/ f0)
Prop4     :  (f1 => f3) => (f0 \/ f1) => (f0 \/ f3)
Prop5     :  f0 == --f0
Prop6     :  -(f0 \/ f1) == (-f0 /\ -f1)
Prop7     :  f0 == f0
Prop8     :  f0 => T
Prop9     :  -(f0 /\ f1) == (-f0 \/ -f1)
Prop10    :  f0 => f0
Prop11    :  f0 => (f0 \/ f1)
Prop12    :  F == -T
Propeq0   :  |-(f0 /\ f1)          = (|-(f0) and |-(f1))
Propeq1   :  |-(f0 /\ f1 => f2)    = |-(f0 => (f1 => f2))
```

```
Propeq2   :  |-(f0 /\ f1 => f2)     = |-(f1 => (f0 => f2))
Propeq3   :  |-((f0 /\ -f1) => f2)  = |-(f0 => (f1 \/ f2))
Propeq4   :  |-(f0 => (f1 \/ f2))   = |-((-f2) => (f0 => f1))
Propeq5   :  |-(-f0 => -f1)         = |-(f1 => f0)
Propeq6   :  |-(f0 => (-f1 => -f2)) = |-(f0 => (f2 => f1))
Propeq7   :  |-((-f0) == (-f1))     = |-(f0 == f1)
Propeq8   :  |-(f0 == f1)           = |-(f1 == f0)
Propeq9   :  |-(f0 => f1)           = |-(-f0 \/ f1)
Propeq10  :  |-(T => f0)            = |-(f0)
Propeq11  :  |-((f0 /\ -f1) => f2)  = |-(f0 => (f2 \/ f1))
Propeq12  :  |-(f0 /\ f1 => f2)     = |-(f0 /\ f1 => (T /\ f2))
Propeq13  :  |-(F)                  = |-(-T)
Propimp0  :  |-(f0) or |-(f1) implies |-(f0 \/ f1)
Propimp1  :  |-(f0=>f1) and |-(f2=>f1) implies |-((f0 \/ f2)=>f1)
Propimp2  :  |-(f0=>f1) and |-(f1=>f0) implies |-(f0==f1)
Propimp3  :  |-(f0==f1) implies (|-(f0=>f1) and |-(f1=>f0))
Propimp4  :  |-(f1==f0) and |-(f2=>f1) implies |-(f2=>f0)
Propimp5  :  |-(f0=>f1) and |-(f0=>f2) implies |-(f0=>(f1 /\ f2))
Propimp6  :  |-(f0=>(f1 /\ f2)) implies (|-(f0=>f1) and |-(f0=>f2))
Propimp7  :  |-(f1==f0) and |-(f2==f1) implies |-(f2==f0)
Propimp8  :  |-(f0==f1) and |-(f2=>-f0) implies |-(f2=>-f1)
Propimp9  :  |-(f0=>f1) and |-(f1=>f2) implies |-(f0=>f2)
Propimp10 :  |-(f0==f1) and |-(f0=>f2) implies |-(f1=>f2)
Propimp11 :  |-(f0==f1) and |-(f2==f3) implies |-((f0 \/ f2)==(f1 \/ f3))
Propimp12 :  |-(f0=>f1) implies |-((f2=>f0)=>(f2=>f1))
Propimp13 :  |-(f0=>(f1=>f2)) and |-(f0=>(f2=>f1)) implies |-(f0=>(f1==f2))
Propimp14 :  |-(f0=>(f1==f2)) implies (|-(f0=>(f1=>f2)) and |-(f0=>(f2=>f1)))
Propimp15 :  |-(f0==f1) implies |-((f0 /\ f2)==(f1 /\ f2))
Propimp16 :  |-(f0==f1) and |-(f2==f3) implies |-((f0 /\ f2)==(f1 /\ f3))
Propimp17 :  |-(f0=>f1) and |-(f2 => f3) implies |-((f0 /\ f2)=>(f1 /\ f3))
Propimpeq1:  |-(f0 == f1)  implies (|-(f0) = |-(f1))

NextChop:                    (O(f0)^f1) == O(f0^f1)
BiChopImpChop:               Bi(f0=>f1)  =>  (f0^f2) => (f1^f2)
BoxChopImpChop:              [](f0=>f1) =>  (f2^f0) => (f2^f1)
LeftChopImpChop:             |-(f0 => f1) implies |-((f0^f2) => (f1^f2))
RightChopImpChop:            |-(f0 => f1) implies |-((f2^f0) => (f2^f1))
OrChopEqv:                   ((f0 \/ f1)^f2) == ((f0^f2) \/ (f1^f2))
ChopOrEqv:                   (f0^(f1 \/ f2)) == ((f0^f1) \/ (f0^f2))
OrChopImpRule:               |-(f0 => (f1 \/ f2)) implies |-(f0^f3 => ((f1^f3) \/ (f2^f3)))
ChopOrImpRule:               |-(f0 => (f1 \/ f2)) implies |-(f3^f0 => ((f3^f1) \/ (f3^f2)))
LeftChopEqvChop:             |-(f0 == f1) implies |-((f0^f2) == (f1^f2))
RightChopEqvChop:            |-(f0 == f1) implies |-((f2^f0) == (f2^f1))
OrChopEqvRule:               |-(f0 == (f1 \/ f2)) implies |-((f0^f3) == ((f1^f3) \/ (f2^f3)))
ChopOrEqvRule:               |-(f1 == (f2 \/ f3)) implies |-((f0^f1) == ((f0^f2) \/ (f0^f3)))
NextImpNext:                 |-(f0 => f1) implies |-(O(f0) => O(f1))
NextImpDist:                 O(f0 => f1) => (O(f0) => O(f1))
AndChopImp:                  ((f0 /\ f1)^f2) => ((f0^f2) /\ (f1^f2))
ChopAndImp:                  (f0^(f1 /\ f2)) => ((f0^f1) /\ (f0^f2))
ChopImpDiamond:              (isfin(f0)^f1) => <> f1
NowImpDiamond:               f1 => <>f1
NextDiamondImpDiamond:       O(<> f1) => <>f1
DiamondNextImpDiamond:       <>(O(f1)) => <>f1
NextEqvNext:                 |-(f0 == f1) implies |-(O(f0) == O(f1))
BoxImpNowAndWeakNext:        []f1 => (f1 /\ wO([]f1))
BoxImpBoxRule:               |-(f0 => f1) implies |-([](f0) => [](f1))
BoxImpDist:                  [](f0 => f1) => ([]f0 =>[]f1)
DiamondEmptyEqvFinite:       (<> empty) == finite
NextAndNextImpNextRule:      |-((f0 /\ f1) => f2) implies |-((O(f0) /\ O(f1)) => O(f2))
NextAndNextEqvNextRule:      |-((f0 /\ f1) ==  f2) implies |-((O(f0) /\ O(f1)) == O(f2))
WeakNextEqvWeakNext:         |-(f0 == f1) implies |-(wO(f0) == wO(f1))
DiamondImpDiamond:           |-(f0 => f1) implies |-(<>(f0) => <>(f1))
DiamondEqvDiamond:           |-(f0 == f1) implies |-(<>(f0) == <>(f1))
BoxEqvBox:                   |-(f0 == f1) implies |-([](f0) == [](f1))
BoxAndBoxImpBoxRule:         |-((f0 /\ f1) => f2) implies |-(([](f0) /\ [](f1)) => [](f2))
BoxAndBoxEqvBoxRule:         |-((f0 /\ f1) == f2) implies |-(([](f0) /\ [](f1)) == [](f2))
BoxAndBoxEqvBoxAnd:          ([](f0) /\ [](f1)) == [](f0 /\ f1)
DiamondOrDiamondEqvDiamondOr: (<>(f0) \/ <>(f1)) == <>(f0 \/ f1)
```

```
BoxIntro:                   |-(f0 => f1) and |-(f0 => wO(f0)) implies |-(f0 => []f1)
DiamondIntro:               |-(f1 => f0) and |-(O(f0) => f0) implies |-(<>f1 => f0)
NextLoop:                   |-(f0 => O(f0)) implies |-(f0 => []f0)
EmptyNextInducta:           |-(finite) and |-(empty => f0) and |-(O(f0) => f0) implies |-(f0)
EmptyNextInductb:           |-(finite) and |-(empty=>(f0 => f1)) and
                            |-(O(f0 => f1) => (f0 => f1)) implies |-(f0 => f1)
FinImpFin:                  |-(f0 => f1) implies |-(fin(f0) => fin(f1))
FinEqvFin:                  |-(f0 == f1) implies |-(fin(f0) == fin(f1))
FinAndFinImpFinRule:        |-((f0 /\ f1) => f2) implies |-((fin(f0) /\ fin(f1)) => fin(f2))
FinAndFinEqvFinRule:        |-((f0 /\ f1) == f2) implies |-((fin(f0) /\ fin(f1)) == fin(f2))
HaltEqvHalt:                |-(f0 == f1) implies |-(halt(f0) == halt(f1))

BiImpDiImpDi:               Bi(f0 => f1) => (Di(f0) => Di(f1))
DiImpDi:                    |-(f0 => f1) implies |-(Di(f0) => Di(f1))
BiImpBiRule:                |-(f0 => f1) implies |-(Bi(f0) => Bi(f1))
DiEqvDi:                    |-(f0 == f1) implies |-(Di(f0) == Di(f1))
NotDiFalse:                 -Di(F)
BiEqvBi:                    |-(f0 == f1) implies |-(Bi(f0) == Bi(f1))
LeftChopChopImpChopRule:    |-((f0^f1) => f1) implies |-((f0^f1^f2) => (f1^f2))
AndChopCommute:             ((f0 /\ f2)^f1) == ((f2 /\ f0)^f1)
ChopAndCommute:             (f0^(f1 /\ f2)) == (f0^(f2 /\ f1))
AndChopAndCommute:          ((f0 /\ f1)^(f2 /\ f3)) == ((f1 /\ f0)^(f3 /\ f2))
ImpDi:                      f0  => Di(f0)
DiState:                    Di(p) == p
StateChop:                  (p^f0) => p
StateChopExportA:           ((p /\ f0)^f1) => p
BiAndChopImport:            (Bi(f0) /\ (f2^f1)) => ((f0 /\ f2)^f1)
StateAndChopImport:         (p /\ (f0^f1)) => (p /\ f0)^f1
StateAndChop:               ((p /\ f0)^f1) == (p /\ (f0^f1))
StateAndDi:                 (p /\ Di(f0)) == Di(p /\ f0)
DiNext:                     Di(O(f0)) == O(Di(f0))
DiNextState:                Di(O(p)) == O(p)
StateImpBiGen:              |-(p => f0) implies |-(p => Bi(f0))
StateAndEmptyChop:          ((p /\  empty)^f0) == (p /\ f0)
StateAndNextChop:           ((p /\ O(f0))^f1) == (p /\ O(f0^f1))
StateAndChopImpChopRule:    |-((p /\ f0) => f2) implies |-((p /\ (f0^f1)) => (f2^f1))
StateImpChopEqvChop:        |-(p => (f0 == f2)) implies |-(p => ((f0^f1) == (f2^f1)))
ChopEqvStateAndChop:        |-(f0 == (p /\ f2)) implies |-((f0^f1) == (p /\ (f2^f1)))
DiIntro:                    f0  => Di(f0)
BiElim:                     Bi(f0) => f0
StateEqvBi:                 p == Bi(p)
DiNotEqvNotBi:              Di(-f0) == -Bi(f0)
DiEqvNotBiNot:              Di(f0) == -Bi(-f0)
BiContraPosImpDist:         Bi(-f1 => -f0) => (Bi(f0) => Bi(f1))
BiImpDist:                  Bi(f0 => f1)  => (Bi(f0) => Bi(f1))
IfChopEqvRule:              |-(f0 == ife(p,f2,f3)) implies |-((f0^f1) == ife(p,(f2^f1), (f3^f1)))
EmptyOrChopEqv:             ((empty \/ f0)^f1) == (f1 \/ (f0^f1))
EmptyOrNextChopEqv:         ((empty \/ O(f0))^f1) == (f1 \/ O(f0^f1))
EmptyOrChopImpRule:         |-(f0 => (empty \/ f2)) implies |-((f0^f1) => (f1 \/ (f2^f1)))
EmptyOrChopEqvRule:         |-(f0 == (empty \/ f2)) implies |-((f0^f1) == (f1 \/ (f2^f1)))
EmptyOrNextChopImpRule:     |-(f0 => (empty \/ O(f2))) implies |-((f0^f1) => (f1 \/ O(f2^f1)))
EmptyOrNextChopEqvRule:     |-(f0 == (empty \/ O(f2))) implies |-((f0^f1) == (f1 \/ O(f2^f1)))
ChopEmptyOrImpRule:         |-(f1 => (empty \/ f2)) implies |-((f0^f1) => (f0 \/ (f0^f2)))
ChopImpChop:                |-(f0 => f2) and |-(f1 => f3) implies |-((f0^f1) => (f2^f3))
ChopEqvChop:                |-(f0 == f2) and |-(f1== f3) implies |-((f0^f1) == (f2^f3))
NotChopEqvYieldsNot:        -(f0^f1) == (f0 |>  -f1)
StateYieldsEqv:             (p  => (f0 |> f1)) == ((p /\ f0) |> f1)
ChopAndNotChopImp:          ((f0^f1) /\ -(f0^f2)) => (f0^(f1 /\ -f2))

ForallSub:                  not member(v1,bound(f1)) and (t(v1)=static implies |-(stable(pe))) and
                            (forall (z:(vr?)): member(z,freeexp(pe)) implies not member(z,bound(f1)))
                            implies  |-(FA(v1,f1) => suform(f1,v1,pe))
ForallImplies:              not member(v1,freeform(f1)) implies |-(FA(v1,f1=>f2) => (f1 => FA(v1,f2)))
ExistsChopRight:            not member(v1,freeform(f2)) implies |-(TE(v1,f1^f2) => (TE(v1,f1)^f2))
ExistsChopLeft:             not member(v1,freeform(f1)) implies |-(TE(v1,f1^f2) => (f1^TE(v1,f2)))

%%%% the rule
ForallGen:                  |-(f0) implies |-(FA(v1,f0))
```