Participants' proceedings, DCC 2004, Barcelona, 27-28 March, 2004.

PSL semantics in higher order logic

Mike Gordon, University of Cambridge Computer Laboratory, 23 February 2004

1. Introduction

In a paper, published in the journal *Formal Aspects of Computing* (FAC) [Gor03][1], we described a deep semantic embedding of Version 1.01 of the Accellera Property Specification Language (PSL) in higher order logic. The main goal of that paper was to demonstrate that mechanised theorem proving can be a useful aid to the validation of the semantics of an industrial design language.

In another paper, presented at CHARME 2003 [GHS03], we showed how mechanised deduction could be applied to a formal encoding of the PSL semantics in higher order logic to generate correct-by-construction tools (a property evaluator, a simulation monitor generator and a model checker). The point of that paper was to show that a formal semantics was not just documentation, but could be executed by special purpose theorem proving scripts.

This document gives more detail than the published papers on how the semantics is represented in the HOL system. It also reflects the (not yet released) Version 1.1 semantics. Some material has been taken from the FAC paper, but the details are updated to correspond to the latest version of PSL.

2. Review of higher order logic, the HOL system and semantic embedding

Higher order logic is an extension of first-order predicate calculus that allows quantification over functions and relations. It is a natural notation for formalising informal set theoretic specifications (indeed, it is usually more natural than formal first-order set theories, like ZF). We hope that the formal logic notation in what follows is sufficiently close to standard informal mathematics that it needs no systematic explanation. In this section we briefly outline some features of the version of higher order logic implemented in the HOL4 system. We refer to this logic as "the HOL logic" or just "HOL".

The HOL logic is built out of *terms* which are of four types: constants, variables, combinations (or function applications) $t_1\ t_2$ and λ-abstractions $\lambda x.\ t$.

The particular set of constants that are available depends on the theory one is working in. The kernel of the HOL logic contains constants `T` and `F` representing truth and falsity, respectively. In the HOL system, new constants can be defined in terms of existing constants using definitional mechanisms that guarantee no new inconsistencies are introduced. Defined constants include numerals (e.g. $0, 1, 2$), strings (e.g. `"a"`, `"b"`, `"ab"`) and logical operators (e.g. $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$). The details of HOL's theory of definition are available elsewhere [GM93].

The simple kernel of four kinds of terms can be extended using syntactic sugar to include all the normal notations of predicate calculus. The extension process consists of defining new constants and then adding syntactic sugar to make terms containing these constants look familiar. For example, constants $\forall$, $\exists$ and `Pair` can be defined and then $\forall x.\ \exists y.\ P(x, y)$ is syntactic sugar for $\forall(\lambda x.\ \exists(\lambda y.\ P\ (\texttt{Pair}\ x\ y)))$, (here the function application `Pair` $x\ y$ means $((\texttt{Pair}\ x)\ y)$, so `Pair` is 'curried'). If P is a function that returns a truth-value (i.e. a predicate), then P can be thought of as a set, and we write $x \in P$ to mean $P(x)$ is true. The term $\lambda x.\ \cdots x \cdots$ corresponds to the set abstraction $\{x \mid \cdots x \cdots\}$ and we will write $\forall x \in P.\ Q(x)$ and $\exists x \in P.\ Q(x)$ to mean $\forall x.\ P(x) \Rightarrow Q(x)$ and $\exists x.\ P(x) \wedge Q(x)$, respectively.

Address: Mike Gordon, University of Cambridge Computer Laboratory, William Gates Building, JJ Thomson Avenue, Cambridge CB3 0FD, U.K. e-mail: mjcg@cl.cam.ac.uk

[1] Draft online at: `http://www.cl.cam.ac.uk/~mjcg/Sugar/facpaper/`.

Higher order logic is typed to avoid inconsistencies.[2] Types are syntactic constructs that denote sets of values. For example, types *bool* and *num* are atomic types in HOL and denote the sets of booleans and natural numbers, respectively. Complex types can be built using type constructors. For example, if ty_1 and ty_2 are types, then $ty_1 \rightarrow ty_2$ denotes the set of functions with domain ty_1 and range ty_2, and $ty_1 \times ty_2$ denotes the Cartesian product of the sets denoted by ty_1 and ty_2. Type constructors are traditionally applied to their arguments using a postfix notation like $(ty_1, \ldots, ty_n)constructor$. The types $ty_1 \rightarrow ty_2$ and $ty_1 \times ty_2$ are just special notations for $(ty_1, ty_2)fun$ and $(ty_1, ty_2)prod$, respectively.

If the types for all the variables and constants in a term t are given, then a type-checking algorithm can determine whether t is well-typed – i.e. every function is applied to an argument of the correct type – and compute a type for t. For example, $\neg 3$ is not well-typed (assuming $\neg$ has type *bool*→*bool* and 3 has type *num*) and would be rejected by type-checking, however, $\neg$`T` is well-typed (assuming `T` has type *bool*) and would be accepted and given type *bool*. Only the well-typed terms are considered meaningful and we write $t : ty$ if term t is well-typed and has type ty. Well-typed terms of type *bool* are the formulas of the HOL logic, thus formulas are a subset of terms: $\forall x.\ \exists y.\ x + 1 < y$ is a term that is a formula, but $x + 1$ is a term (of type *num*) that is not a formula. The HOL logic kernel only has two types and one type constructor: type *bool* of booleans, an infinite type *ind* of 'individuals' and the function type constructor $\rightarrow$. Other types and type constructors can be defined in terms of these [GM93]. For example, the type *num* of numbers is defined as a subset of the primitive type *ind*, and the Cartesian product constructor $\times$ can be defined in terms of $\rightarrow$. Families of terms can be created by using type variables. For example, if variable x is assigned the type α, where α is a type variable, then $\lambda x.\ x$ has type $\alpha \rightarrow \alpha$ and is a family of identity functions with an instance $\lambda x : ty.\ x$ for each type ty.

2.1. HOL system notation

Input to the HOL system uses ASCII characters. The table below shows some common idioms, including those that are used in this paper.

Standard notation	HOL notation	Description
true	`T`	truth
false	`F`	falsity
$\neg t$	`~`t	negation
$t_1 \wedge t_2$	t_1 `/\` t_2	conjunction
$t_1 \vee t_2$	t_1 `\/` t_2	disjunction
$t_1 \Rightarrow t_2$	t_1 `==>` t_2	implication
$\forall x.P(x)$	`!`x`.`$P(x)$	universal quantification
$\exists x.P(x)$	`?`x`.`$P(x)$	existential quantification
$p \in s$	p `IN` s	set membership
`[`$0..n$`)`	`LESS` n	set of natural numbers less than n
$\forall x \in s.\ P(x)$	`!`x`::s.` $P(x)$	universal quantification restricted to s
$\exists x \in s.\ P(x)$	`?`x`::s.` $P(x)$	existential quantification restricted to s
$\forall x \in$ `[`$0..n$`)`. $P(x)$	`!`x`::LESS` n`.` $P(x)$	universal quantification restricted to numbers less than n
$\exists x \in$ `[`$0..n$`)`. $P(x)$	`?`x`::LESS` n`.` $P(x)$	existential quantification restricted to numbers less than n
ϵ	`[]`	empty list
x	`[`x`]`	list with one element (singleton)
$l_1 l_2$	l_1 `<>` l_2	list concatenation (append)
$ty_1 \times ty_2$	ty_1 `#` ty_2	Cartesian product of types ty_1 and ty_2
$ty_1 \rightarrow ty_2$	ty_1 `-->` ty_2	type of functions from ty_1 to ty_2

To enable an easy comparison with the informal presentation in the PSL Language Reference Manual (LRM), we include snippets from the LRM in framed boxes[3]

[2] Russell's paradox can be formulated as: $(\lambda x.\ \neg(x\ x))\ (\lambda x.\ \neg(x\ x)) = \neg((\lambda x.\ \neg(x\ x))\ (\lambda x.\ \neg(x\ x)))$.
[3] Thanks to Dana Fisman for supplying LaTeX source of the draft LRM. Not that as we are using a different style file for typesetting, the appearance of the material in the boxes may be formatted here differently from how the text will appear in the forthcoming LRM.

2.2. Representing letters and words in HOL

In LRM Version 1.1 (Section B.2.1) we find:

> The semantics of FL is defined with respect to finite and infinite words over $\Sigma = 2^P \cup \{\top, \bot\}$.

Members of Σ are called letters and to represent them in HOL we define a type `('a)letter`, where the parametrisation on a type variable `'a` is so that different sets P of atomic propositions can be 'plugged-in' by instantiating `'a` to a type representing P.

A data-type definition has the form `Hol_datatype` `‘<`*description of type*`>‘`. The following input to HOL defines a new type `('a)letter` together with three constants (which are separated by "`|`").

```
Hol_datatype ‘letter = TOP | BOTTOM | STATE of ('a -> bool)‘
```

Note that the syntax used for declaring data-types in the HOL system logic requires the type name without any parameters on the left hand side (i.e. `letter` rather than `('a)letter`). The presence of the single free type variable `'a` in the right hand side causes a unary type operator to be defined.

The constants `TOP` and `BOTTOM` are distinct values of type `('a)letter`. The constant `STATE` is a function taking an argument of the type shown after the "`of`" and returning a result of type `('a)letter`. Thus the effect of executing the data-type definition is to define a new type `('a)letter` together with the following constants.

```
TOP     : ('a)letter
BOTTOM  : ('a)letter
STATE   : ('a → bool) → ('a)letter
```

The argument to `STATE` is the characteristic function of a set of atomic propositions. When HOL performs such a definition it automatically proves a standard set of useful theorems about the type and the constants defined on it (e.g. `~(TOP = BOTTOM)`, which represents $\neg(\top = \bot)$).

The PSL LRM continues:

> We denote a letter from Σ by ℓ and an empty, finite, or infinite word from Σ by u, v, or w (possibly with subscripts).

Finite paths can be represented by a built-in type `list` of finite lists. Infinite paths can be represented as functions from natural numbers (type `num`). Thus to represent paths in HOL we define a disjoint union type:

```
Hol_datatype
 ‘path = FINITE of ('s list) | INFINITE of (num -> 's)‘
```

This defines a unary type operator `path`. A type (ty)`path` represents paths whose elements are of type ty.

Next the PSL LRM says:

> We denote the length of word v as $|v|$. An empty word $v = \epsilon$ has length 0, a finite word $v = (\ell_0\ell_1\ell_2\cdots\ell_n)$ has length $n+1$, and an infinite word has length ∞.

The length of a path is thus either a natural number or is ∞. To model this we define a type `xnum` of extended natural numbers. Comments in HOL are enclosed between `(*` and `*)`.

```
Hol_datatype
 ‘xnum = INFINITY                          (* length of an infinite path  *)
       | XNUM of num‘                      (* length of a finite path     *)
```

This defines the type `xnum` together with the following constants.

```
INFINITY : xnum
XNUM     : num → xnum
```

The length of a path can now be defined in the HOL logic by defining a constant `LENGTH : ('a)path` $\rightarrow$ `xnum`. The function `list$LENGTH`, which occurs below, is the pre-existing length function on finite lists.

```
Define `(LENGTH(FINITE l)   = XNUM(list$LENGTH l))
        /\
        (LENGTH(INFINITE p) = INFINITY)`
```

This definition overloads the name `LENGTH` so it now can be applied both to lists and to paths.

Continuing with B2.1 of the PSL LRM:

> We use i, j, and k to denote non-negative integers. We denote the i^{th} letter of v by v^{i-1} (since counting of letters starts at zero). We denote by $v^{i..}$ the suffix of v starting at v^i. That is, for every $i < |v|$, $v^{i..} = v^i v^{i+1} \cdots v^n$ or $v^{i..} = v^i v^{i+1} \cdots$. We denote by $v^{i..j}$ the finite sequence of letters starting from v^i and ending in v^j. That is, for $j \geq i$, $v^{i..j} = v^i v^{i+1} \cdots v^j$ and for $j < i$, $v^{i..j} = \epsilon$. We use ℓ^ω to denote an infinite-length word, each letter of which is ℓ.
> We use $\overline{v}$ to denote the word obtained by replacing every $\top$ with a $\bot$ and vice versa. We call $\overline{v}$ the *complement* of v.

These operations are straightforward to define by 'functional programming' in the HOL logic. We do not give the definitions here, but show in the table below the PSL notation and corresponding HOL representation.

PSL Notation	HOL representation	Description
∞	`INFINITY`	infinity
ϵ	`[]`	empty path
$\top^\omega$	`TOP_OMEGA`	infinite repetition of $\top$
$\|v\|$	`LENGTH` v	length of a path
v^i	`ELEM` v i	i+t^{th} letter of v
$v^{i..}$	`RESTN` v i	suffix of v starting at v^i
$v^{i..j}$	`SEL` v (i,j)	sequence starting at v^i and ending at v^j
$\overline{v}$	`COMPLEMENT` v	complement of v (swap $\top$s and $\bot$s)

3. Representing syntax in higher order logic

PSL has four classes of constructs: boolean expressions, Sequential Extended Regular Expressions (SEREs), Foundation Language (FL) formulas and Optional Branching Extension (OBE) formulas. The OBE is ignored here, though for PSL Version 1.01 its semantics in HOL appears in the FAC paper.

Although the syntax of boolean expressions is not explicitly defined, it says in Section B.1 of the LRM:

> The logic Accellera PSL is defined with respect to a non-empty set of atomic propositions P and a given set of boolean expressions B over P. We assume two designated boolean expression *true* and *false* belong to B.

In addition, in LRM B.2.1 the semantics of boolean expressions $\neg b$ and $b_1 \wedge b_2$ are defined, so we include these as primitives too.

Abstract syntax is represented in HOL by defining a data-type whose operations are the constructors.

For boolean expressions, a data-type `bexp` is defined. Since atomic propositions are boolean expressions, we parameterise the type of boolean expressions on a type variable `'a` that can be subsequently instantiated to a particular type representing the set P of atomic propositions. If `aprop` is such a type, then the type of terms representing boolean expressions is `(aprop)bexp`. Thus `bexp` is a unary type constructor.

When a constructors is to take n arguments, where $n > 1$, one writes "`of` ty_1 `#` $\cdots$ `#` ty_n" after a constructor name in the data-type declaration, where ty_1, ..., ty_n are the types of the arguments.

The input to the HOL system to define `bexp` is:

```
Hol_datatype
 `bexp = B_PROP   of 'a                     (* atomic proposition       *)
       | B_TRUE                             (* true                     *)
       | B_FALSE                            (* false                    *)
       | B_NOT    of bexp                   (* negation                 *)
       | B_AND    of bexp # bexp`           (* conjunction              *)
```

This defines a new unary type constructor *bexp* and constants:

```
B_PROP  : 'a → ('a)bexp
B_TRUE  : ('a)bexp
B_FALSE : ('a)bexp
B_NOT   : ('a)bexp → ('a)bexp
B_AND   : ('a)bexp × ('a)bexp → ('a)bexp
```

The prefix `B_` indicates a boolean expression constructor.

If atomic propositions are taken to be strings, then the boolean expression $x \wedge \neg y$ would be represented by the term `B_AND(B_PROP "x", B_NOT(B_PROP "y"))` which has the type `(string)bexp`.

The syntax of SEREs is described in the LRM by:

Definition 1 (Sequential Extended Regular Expressions (SEREs)).

– Every boolean expression $b \in B$ is a SERE.
– If r, r_1, and r_2 are SEREs, and c is a boolean expression, then the following are SEREs:
  • $\{r\}$ • $r_1 \; ; \; r_2$ • $r_1 : r_2$ • $r_1 \mid r_2$
  • r_1 && r_2 • $[*0]$ • $r[*]$ • r@c

This is represented in HOL by defining a data-type `sere` by (the prefix `S_` indicates a SERE constructor):

```
Hol_datatype
 `sere = S_BOOL        of 'a bexp             (* boolean expression       *)
       | S_CAT         of sere # sere         (* r1 ;  r2                 *)
       | S_FUSION      of sere # sere         (* r1 :  r2                 *)
       | S_OR          of sere # sere         (* r1 |  r2                 *)
       | S_AND         of sere # sere         (* r1 && r2                 *)
       | S_EMPTY                              (* [*0]                     *)
       | S_REPEAT      of sere                (* r[*]                     *)
       | S_CLOCK       of sere # 'a bexp`     (* r@c                      *)
```

This defines a unary type operator `sere` (the need for parametrisation is inferred from the free type variable `'a` in the right hand side of the definition).

The syntax of FL formulas is defined in the LRM by (the prefix `F_` indicates an FL formula constructor):

Definition 2 (Formulas of the Foundation Language (FL formulas)).

– If b is a boolean expression then both b and $b!$ are FL formulas.
– If φ and ψ are FL formulas, r, r_1, r_2 are SEREs, and b a boolean expression, then the following are FL formulas:
  • (φ) • $\neg\varphi$ • $\varphi \wedge \psi$ • $r!$ • r
  • $X!\ \varphi$ • $[\varphi\ U\ \psi]$ • φ abort b • $r \mapsto \varphi$ • φ@b

This is represented in HOL by defining a data-type `fl` by:

```
Hol_datatype
 `fl = F_STRONG_BOOL  of 'a bexp              (* b!                          *)
     | F_WEAK_BOOL    of 'a bexp              (* b                           *)
     | F_NOT          of fl                   (* not f                       *)
     | F_AND          of fl # fl              (* f1 and f2                   *)
     | F_STRONG_SERE  of 'a sere              (* r!                          *)
     | F_WEAK_SERE    of 'a sere              (* r                           *)
     | F_NEXT         of fl                   (* X! f                        *)
     | F_UNTIL        of fl # fl              (* [f1 U f2]                   *)
     | F_ABORT        of fl # 'a bexp         (* f abort b                   *)
     | F_CLOCK        of fl # 'a bexp         (* f@b                         *)
     | F_SUFFIX_IMP   of 'a sere # fl`        (* r  |-> f                    *)
```

This defines a unary type operator `fl`.

4. Formal semantics in higher order logic

In this section we give the semantics that is expected to be released in the forthcoming LRM for Accellera PSL Version 1.1. We then show its representation in HOL, both pretty printed and in raw ASCII form.

4.1. Boolean expressions in PSL

The semantics of boolean expressions is described in the LRM as follows:

The semantics of boolean expression is assumed to be given as a relation $\Vdash\ \subseteq \Sigma \times B$ relating letters in Σ with boolean expressions in B. If $(\ell, b) \in\ \Vdash$ we say that the letter ℓ *satisfies* the boolean expression b and denote it $\ell \Vdash b$. We assume the two special letters $\top$ and $\bot$ behave as follows: for every boolean expression b, $\top \Vdash b$ and $\bot \nVdash b$. We assume that otherwise the boolean relation $\Vdash$ behaves in the usual manner. In particular, that for every letter $\ell \in 2^P$, atomic proposition $p \in P$ and boolean expressions $b, b_1, b_2 \in B$ (i) $\ell \Vdash p$ iff $p \in \ell$, (ii) $\ell \Vdash \neg b$ iff $\ell \nVdash b$, and (iii) $\ell \Vdash$ *true* and $\ell \nVdash$ *false*. Finally, we assume that for every letter $\ell \in \Sigma$, $\ell \Vdash b_1 \wedge b_2$ iff $\ell \Vdash b_1$ and $\ell \Vdash b_2$.

The semantics of boolean expressions is represented in HOL by defining a new constant corresponding to a semantic function B_SEM : `('a→bool)→('a)bexp→bool` such that B_SEM l b is true iff b is true with respect to letter l. The actual input to HOL to define S_SEM is:

```
Define
 `(B_SEM TOP b = T)
  /\
  (B_SEM BOTTOM b = F)
  /\
  (B_SEM (STATE s) (B_PROP p) = p IN s)
  /\
  (B_SEM (STATE s) B_TRUE = T)
  /\
  (B_SEM (STATE s) B_FALSE = F)
  /\
  (B_SEM (STATE s) (B_NOT b) = ~(B_SEM (STATE s) b))
  /\
  (B_SEM (STATE s) (B_AND(b1,b2)) = B_SEM (STATE s) b1 /\ B_SEM (STATE s) b2)`
```

If `B_SEM l b` is pretty printed as $l \Vdash b$, then the semantics above pretty prints as:

$$
\begin{array}{l}
(\top \Vdash b \ = \ \mathrm{T}) \\
\wedge \\
(\bot \Vdash b \ = \ \mathrm{F}) \\
\wedge \\
(s \Vdash p \ = \ p \in s) \\
\wedge \\
(s \Vdash \mathrm{T} \ = \ \mathrm{T}) \\
\wedge \\
(s \Vdash \mathrm{F} \ = \ \mathrm{F}) \\
\wedge \\
(s \Vdash \neg b \ = \ \neg(s \Vdash b)) \\
\wedge \\
(s \Vdash b_1 \wedge b_2 \ = \ s \Vdash b_1 \wedge s \Vdash b_2)
\end{array}
$$

Pretty-printing introduces potentially confusing overloading: the occurrence of $\neg$ in $\neg b$ is part of the boolean expression syntax of PSL, but the occurrence in $\neg(l \models b)$ is negation in higher order logic. Similarly $\wedge$ is overloaded: the occurrence in $b_1 \wedge b_2$ is part of the boolean expression syntax, but the other occurrences are conjunction in higher order logic.

4.2. Extended Regular Expressions (SEREs)

The unclocked semantics (B.2.1.1.1 of the LRM) is shown in the next box:

Unclocked SEREs are defined over finite words from the alphabet Σ. The notation $v \models r$, where r is a SERE and v a finite word means that v *models tightly* r. The semantics of unclocked SEREs are defined as follows, where b denotes a boolean expression, and r, r_1, and r_2 denote unclocked SEREs.

- $v \models \{r\} \Longleftrightarrow v \models r$
- $v \models b \Longleftrightarrow |v| = 1$ and $v^0 \Vdash b$
- $v \models r_1 \ ; \ r_2 \Longleftrightarrow \exists v_1, v_2$ s.t. $v = v_1 v_2$, $v_1 \models r_1$, and $v_2 \models r_2$
- $v \models r_1 : r_2 \Longleftrightarrow \exists v_1, v_2$, and ℓ s.t. $v = v_1 \ell v_2$, $v_1 \ell \models r_1$, and $\ell v_2 \models r_2$
- $v \models r_1 \mid r_2 \Longleftrightarrow v \models r_1$ or $v \models r_2$
- $v \models r_1 \ \&\& \ r_2 \Longleftrightarrow v \models r_1$ and $v \models r_2$
- $v \models [*0] \Longleftrightarrow v = \epsilon$
- $v \models r[*] \Longleftrightarrow$ either $v \models [*0]$ or $\exists v_1, v_2$ s.t. $v_1 \neq \epsilon$, $v = v_1 v_2$, $v_1 \models r$ and $v_2 \models r[*]$

The pretty-printed HOL representation of this is:

$$
\begin{array}{l}
(v \models b \ = \ (|v| = 1) \wedge v^0 \Vdash b) \\
\wedge \\
(v \models r_1 ; r_2 \ = \ \exists v_1 v_2 .\ (v = v_1 v_2) \wedge v_1 \models r_1 \wedge v_2 \models r_2) \\
\wedge \\
(v \models r_1 : r_2 \ = \ \exists v_1 v_2 l .\ (v = v_1 [l] v_2) \wedge v_1 [l] \models r_1 \wedge [l] v_2 \models r_2) \\
\wedge \\
(v \models r_1 \mid r_2 \ = \ v \models r_1 \vee v \models r_2) \\
\wedge \\
(v \models r_1 \&\& r_2 = v \models r_1 \wedge v \models r_2) \\
\wedge \\
(v \models [*0] \ = \ (v = \epsilon)) \\
\wedge \\
(v \models r[*] \ = \ v \models [*0] \vee \exists v_1 v_2 .\ \neg(v = \epsilon) \wedge (v = v_1 v_2) \wedge v_1 \models r \wedge v_2 \models r[*])
\end{array}
$$

The raw HOL is (we omit the `Define` and enclosing quotes):

```
(US_SEM v (S_BOOL b) = (LENGTH v = 1) /\ B_SEM (ELEM v 0) b)
/\
(US_SEM v (S_CAT(r1,r2)) = ?v1 v2. (v = v1 <> v2) /\ US_SEM v1 r1 /\ US_SEM v2 r2)
/\
(US_SEM v (S_FUSION(r1,r2)) =
  ?v1 v2 l. (v = v1 <> [l] <> v2) /\ US_SEM (v1<>[l]) r1 /\ US_SEM ([l]<>v2) r2)
/\
(US_SEM v (S_OR(r1,r2)) = US_SEM v r1 \/ US_SEM v r2)
/\
(US_SEM v (S_AND(r1,r2)) = US_SEM v r1 /\ US_SEM v r2)
/\
(US_SEM v S_EMPTY = (v = []))
/\
(US_SEM v (S_REPEAT r) =
  US_SEM v S_EMPTY \/
   ?v1 v2. ~(v=[]) /\ (v = v1 <> v2) /\ US_SEM v1 r /\ US_SEM v2 (S_REPEAT r))
```

The clocked semantics (B.2.1.2.1 of the LRM) is more complex.

We say that finite word v *is a clock tick of* c iff $|v| > 0$ and $v^{|v|-1} \Vdash c$ and for every natural number $i < |v| - 1$, $v^i \Vdash \neg c$.

This is formalised by defining a constant: $\mathsf{ClockTick}(v, c) = |v| > 0 \wedge v^{|v|-1} \Vdash c \wedge \forall i \in [0..|v|-1).\ v^i \Vdash \neg c$.

Clocked SEREs are defined over finite words from the alphabet Σ and a boolean expression that serves as the clock context. The notation $v \overset{c}{\models} r$, where r is a SERE and c is a boolean expression, means that v *models tightly* r *in context of clock* c. The semantics of clocked SEREs are defined as follows, where b, c, and c_1 denote boolean expressions, r, r_1, and r_2 denote clocked SEREs.

– $v \overset{c}{\models} \{r\} \iff v \overset{c}{\models} r$

– $v \overset{c}{\models} b \iff v$ is a clock tick of c and $v^{|v|-1} \Vdash b$

– $v \overset{c}{\models} r_1 \ ;\ r_2 \iff \exists v_1, v_2$ s.t. $v = v_1 v_2$, $v_1 \overset{c}{\models} r_1$, and $v_2 \overset{c}{\models} r_2$

– $v \overset{c}{\models} r_1 : r_2 \iff \exists v_1, v_2$, and ℓ s.t. $v = v_1 \ell v_2$, $v_1 \ell \overset{c}{\models} r_1$, and $\ell v_2 \overset{c}{\models} r_2$

– $v \overset{c}{\models} r_1 \mid r_2 \iff v \overset{c}{\models} r_1$ or $v \overset{c}{\models} r_2$

– $v \overset{c}{\models} r_1 \mathbin{\&\&} r_2 \iff v \overset{c}{\models} r_1$ and $v \overset{c}{\models} r_2$

– $v \overset{c}{\models} [*0] \iff v = \epsilon$

– $v \overset{c}{\models} r[*] \iff$ either $v \overset{c}{\models} [*0]$ or $\exists v_1, v_2$ s.t. $v_1 \neq \epsilon$, $v = v_1 v_2$, $v_1 \overset{c}{\models} r$ and $v_2 \overset{c}{\models} r[*]$

– $v \overset{c}{\models} r@c_1 \iff v \overset{c_1}{\models} r$

The HOL representation of this semantics of SEREs is defined by a semantic function `S_SEM` such that `S_SEM` $w\ c\ r$ is true iff word w is in the language recognised by the extended regular expression r when the clock context (i.e. current clock) is c. The HOL term `S_SEM` $w\ c\ r$ is pretty-printed as $w \overset{c}{\models} r$.

$$
\begin{array}{l}
(v \overset{c}{\models} b \ = \ \mathsf{ClockTick}(v, c) \wedge v^{|v|-1} \Vdash b) \\
\wedge \\
(v \overset{c}{\models} r_1 ; r_2 \ = \ \exists v_1 v_2 .\ (v = v_1 v_2) \wedge v_1 \overset{c}{\models} r_1 \wedge v_2 \overset{c}{\models} r_2) \\
\wedge \\
(v \overset{c}{\models} r_1 : r_2 \ = \ \exists v_1 v_2 l.\ (v = v_1 [l] v_2) \wedge v_1 [l] \overset{c}{\models} r_1 \wedge [l] v_2 \overset{c}{\models} r_2) \\
\wedge \\
(v \overset{c}{\models} r_1 \mid r_2 \ = \ v \overset{c}{\models} r_1 \vee v \overset{c}{\models} r_2) \\
\wedge
\end{array}
$$

$$
\begin{array}{l}
(v \overset{c}{\models} r_1 \&\& r_2 \;=\; v \overset{c}{\models} r_1 \wedge v \overset{c}{\models} r_2) \\
\wedge \\
(v \overset{c}{\models} [*0] \;=\; (v = \epsilon)) \\
\wedge \\
(v \overset{c}{\models} r[*] \;=\; v \overset{c}{\models} [*0] \vee \exists v_1 v_2.\ \neg(v = \epsilon) \wedge (v = v_1 v_2) \wedge v_1 \overset{c}{\models} r \wedge v_2 \overset{c}{\models} r[*]) \\
\wedge \\
(v \overset{c}{\models} r@c_1 \;=\; v \overset{c_1}{\models} r)
\end{array}
$$

The raw HOL input is

```
(S_SEM v c (S_BOOL b) = CLOCK_TICK v c /\ B_SEM (ELEM v (LENGTH v - 1)) b)
/\
(S_SEM v c (S_CAT(r1,r2)) = ?v1 v2. (v = v1 <> v2) /\ S_SEM v1 c r1 /\ S_SEM v2 c r2)
/\
(S_SEM v c (S_FUSION(r1,r2)) =
  ?v1 v2 l. (v = v1 <> [l] <> v2) /\ S_SEM (v1<>[l]) c r1 /\ S_SEM ([l]<>v2) c r2)
/\
(S_SEM v c (S_OR(r1,r2)) = S_SEM v c r1 \/ S_SEM v c r2)
/\
(S_SEM v c (S_AND(r1,r2)) = S_SEM v c r1 /\ S_SEM v c r2)
/\
(S_SEM v c S_EMPTY = (v = []))
/\
(S_SEM v c (S_REPEAT r) =
  S_SEM v c S_EMPTY
  \/ ?v1 v2. ~(v=[]) /\ (v = v1 <> v2) /\ S_SEM v1 c r /\ S_SEM v2 c (S_REPEAT r))
/\
(S_SEM v c (S_CLOCK(r,c1)) = S_SEM v c1 r)
```

4.3. Foundation Language (FL)

FL combines standard LTL notation with a less standard abort operation and some constructs using SEREs. The abstract syntax from B.1 of the LRM is:

The unclocked semantics from B.2.1.1.2 of the LRM is:

We refer to a formula of FL with no @ operator as an *unclocked formula.* Let v be a finite or infinite word, b be a boolean expression, r, r_1, r_2 unclocked SEREs, and φ, ψ unclocked FL formulas. We use $\models$ to define the semantics of unclocked FL formulas: If $v \models \varphi$ we say that v *models* (or *satisfies*) φ.

1. $v \models (\varphi) \iff v \models \varphi$
2. $v \models \neg\varphi \iff \overline{v} \not\models \varphi$
3. $v \models \varphi \wedge \psi \iff v \models \varphi$ and $v \models \psi$
4. $v \models b! \iff |v| > 0$ and $v^0 \Vdash b$
5. $v \models b \iff |v| = 0$ or $v^0 \Vdash b$
6. $v \models r! \iff \exists j < |v|$ s.t. $v^{0..j} \models r$
7. $v \models r \iff \forall j < |v|,\ v^{0..j}\top^\omega \models r!$
8. $v \models X!\ \varphi \iff |v| > 1$ and $v^{1..} \models \varphi$
9. $v \models [\varphi U \psi] \iff \exists k < |v|$ s.t. $v^{k..} \models \psi$, and $\forall j < k,\ v^{j..} \models \varphi$
10. $v \models \varphi$ abort $b \iff$ either $v \models \varphi$ or $\exists j < |v|$ s.t. $v^j \Vdash b$ and $v^{0..j-1}\top^\omega \models \varphi$
11. $v \models r \mapsto \varphi \iff \forall j < |v|$ s.t. $\overline{v}^{0..j} \models r$, $v^{j..} \models \varphi$

The pretty-printed HOL version of this is:

$$
\begin{array}{l}
(v \models \neg f \ = \ \neg(\overline{v} \models f)) \\
\wedge \\
(v \models f_1 \wedge f_2 \ = \ v \models f_1 \wedge v \models f_2) \\
\wedge \\
(v \models b! \ = \ (|v| > 0) \wedge v^0 \Vdash b) \\
\wedge \\
(v \models b \ = \ (|v| = 0) \vee v^0 \Vdash b) \\
\wedge \\
(v \models r! \ = \ \exists j \in [0..|v|).\ v^{0..j} \mathrel{|}\!\equiv r) \\
\wedge \\
(v \models r \ = \ \forall j \in [0..|v|).\ v^{0..j}\top^{\omega} \models r!) \\
\wedge \\
(v \models X!\,f \ = \ |v| > 1 \wedge v^{1..} \models f) \\
\wedge \\
(v \models [f_1\ U\ f_2] \ = \ \exists k \in [0..|v|).\ v^{k..} \models f_2 \wedge \forall j \in [0..k).\ v^{j..} \models f_1) \\
\wedge \\
(v \models f \ \mathsf{abort}\ b \ = \ v \models f \vee \exists j \in [0..|v|).\ v^{j} \Vdash b \wedge v^{0..j-1}\top^{\omega} \models f) \\
\wedge \\
(v \models r \mapsto f \ = \ \forall j \in [0..|v|).\ \overline{v}^{0..j} \mathrel{|}\!\equiv r \Rightarrow v^{j..} \models f)
\end{array}
$$

The raw HOL is

```
(UF_SEM v (F_NOT f) = ~(UF_SEM (COMPLEMENT v) f))
/\
(UF_SEM v (F_AND(f1,f2)) = UF_SEM v f1 /\ UF_SEM v f2)
/\
(UF_SEM v (F_STRONG_BOOL b) = (LENGTH v > 0) /\ B_SEM (ELEM v 0) b)
/\
(UF_SEM v (F_WEAK_BOOL b) = (LENGTH v = XNUM 0) \/ B_SEM (ELEM v 0) b)
/\
(UF_SEM v (F_STRONG_SERE r) = ?j :: LESS(LENGTH v). US_SEM (SEL v (0,j)) r)
/\
(UF_SEM v (F_WEAK_SERE r) =
  !j :: LESS(LENGTH v).
   UF_SEM (CAT(SEL v (0,j),TOP_OMEGA)) (F_STRONG_SERE r))
/\
(UF_SEM v (F_NEXT f) = LENGTH v > 1 /\ UF_SEM (RESTN v 1) f)
/\
(UF_SEM v (F_UNTIL(f1,f2)) =
  ?k :: LESS(LENGTH v).
    UF_SEM (RESTN v k) f2 /\ !j :: LESS k. UF_SEM (RESTN v j) f1)
/\
(UF_SEM v (F_ABORT (f,b)) =
  UF_SEM v f
  \/
  ?j :: LESS(LENGTH v).
     B_SEM (ELEM v j) b /\ UF_SEM (CAT(SEL v (0,j-1),TOP_OMEGA)) f)
/\
(UF_SEM v (F_SUFFIX_IMP(r,f)) =
  !j :: LESS(LENGTH v).
    US_SEM (SEL (COMPLEMENT v) (0,j)) r ==> UF_SEM (RESTN v j) f)
```

The clocked semantics from B.2.1.2.2 of the LRM is:

The semantics of (clocked) FL formulas is defined with respect to finite/infinite words over Σ and a boolean expression c which serves as the clock context. Let v be a finite or infinite word, b, c, c_1 boolean expressions, r, r_1, r_2 SEREs, and φ, ψ FL formulas. We use $\overset{c}{\models}$ to define the semantics of FL formulas. If $v \overset{c}{\models} \varphi$ we say that v *models* (or *satisfies*) φ *in the context of clock c*.

1. $v \overset{c}{\models} (\varphi) \iff v \overset{c}{\models} \varphi$
2. $v \overset{c}{\models} \neg\varphi \iff \overline{v} \overset{c}{\not\models} \varphi$
3. $v \overset{c}{\models} \varphi \wedge \psi \iff v \overset{c}{\models} \varphi$ and $v \overset{c}{\models} \psi$
4. $v \overset{c}{\models} b! \iff \exists j < |v|$ s.t. $v^{0..j}$ is a clock tick of c and $v^j \Vdash b$
5. $v \overset{c}{\models} b \iff \forall j < |v|$ s.t. $\overline{v}^{0..j}$ is a clock tick of c, $v^j \Vdash b$
6. $v \overset{c}{\models} r! \iff \exists j < |v|$ s.t. $v^{0..j} \overset{c}{\models\!\!\models} r$
7. $v \overset{c}{\models} r \iff \forall j < |v|$, $v^{0..j}\top^\omega \overset{c}{\models} r!$
8. $v \overset{c}{\models} X!\ f \iff \exists j < k < |v|$ s.t. $v^{0..j}$ and $v^{j+1..k}$ are clock ticks of c and $v^{k..} \overset{c}{\models} f$
9. $v \overset{c}{\models} [\varphi U \psi] \iff \exists k < |v|$ s.t. $v^k \Vdash c$, $v^{k..} \overset{c}{\models} \psi$, and $\forall j < k$ s.t. $\overline{v}^j \Vdash c$, $v^{j..} \overset{c}{\models} \varphi$
10. $v \overset{c}{\models} \varphi\ \mathsf{abort}\ b \iff$ either $v \overset{c}{\models} \varphi$ or $\exists j < |v|$ s.t. $v^j \Vdash b$ and $v^{0..j-1}\top^\omega \overset{c}{\models} \varphi$
11. $v \overset{c}{\models} r \mapsto \varphi \iff \forall j < |v|$ s.t. $\overline{v}^{0..j} \overset{c}{\models\!\!\models} r$, $v^{j..} \overset{c}{\models} \varphi$
12. $v \overset{c}{\models} \varphi @ c_1 \iff v \overset{c_1}{\models} \varphi$

The HOL semantics is specified by defining a semantic function `F_SEM` such that `F_SEM` $w\ c\ f$ means FL formula f is true of path w with current clock c.
The HOL term `F_SEM` $v\ c\ f$ is pretty printed as $v \overset{c}{\models} f$.

$$
\begin{array}{l}
(v \overset{c}{\models} \neg f \;=\; \neg(\overline{v} \overset{c}{\models} f))\\
\wedge\\
(v \overset{c}{\models} f_1 \wedge f_2 \;=\; v \overset{c}{\models} f_1 \wedge v \overset{c}{\models} f_2)\\
\wedge\\
(v \overset{c}{\models} b! \;=\; \exists j \in [0..|v|).\ \mathsf{ClockTick}(v^{0..j}, c) \wedge v^j \Vdash b)\\
\wedge\\
(v \overset{c}{\models} b \;=\; \forall j \in [0..|v|).\ \mathsf{ClockTick}(\overline{v}^{0..j}, c) \Rightarrow v^j \Vdash b)\\
\wedge\\
(v \overset{c}{\models} r! \;=\; \exists j \in [0..|v|).\ v^{0..j} \overset{c}{\models\!\!\models} r)\\
\wedge\\
(v \overset{c}{\models} r \;=\; \forall j \in [0..|v|).\ v^{0..j}\top^\omega \overset{c}{\models} r!)\\
\wedge\\
(v \overset{c}{\models} X!\,f \;=\; \exists jk \in [0..|v|).\ j < k \wedge \mathsf{ClockTick}(v^{0..j}, c) \wedge \mathsf{ClockTick}(v^{j+1..k}, c) \wedge v^{k..} \overset{c}{\models} f)\\
\wedge\\
(v \overset{c}{\models} [f_1\ U\ f_2] \;=\; \exists k \in [0..|v|).\ v^k \Vdash c \wedge v^{k..} \overset{c}{\models} f_2 \wedge \forall j \in [0..k).\ \overline{v}^j \Vdash c \Rightarrow v^{j..} \overset{c}{\models} f_1)\\
\wedge\\
(v \overset{c}{\models} f\ \mathsf{abort}\ b \;=\; v \overset{c}{\models} f \vee \exists j \in [0..|v|).\ v^j \Vdash b \wedge v^{0..j-1}\top^\omega \overset{c}{\models} f)\\
\wedge\\
(v \overset{c}{\models} f @ c_1 \;=\; v \overset{c_1}{\models} f)\\
\wedge\\
(v \overset{c}{\models} r \mapsto f \;=\; \forall j \in [0..|v|).\overline{v}^{0..j} \overset{c}{\models\!\!\models} r \Rightarrow v^{j..} \overset{c}{\models} f)
\end{array}
$$

The raw HOL is:

```
(F_SEM v c (F_NOT f) = ~(F_SEM (COMPLEMENT v) c f))
/\
(F_SEM v c (F_AND(f1,f2)) = F_SEM v c f1 /\ F_SEM v c f2)
/\
(F_SEM v c (F_STRONG_BOOL b) =
  ?j :: LESS(LENGTH v). CLOCK_TICK (SEL v (0,j)) c /\ B_SEM (ELEM v j) b)
/\
(F_SEM v c (F_WEAK_BOOL b) =
  !j :: LESS(LENGTH v). CLOCK_TICK (SEL (COMPLEMENT v) (0,j)) c ==> B_SEM (ELEM v j) b)
/\
(F_SEM v c (F_STRONG_SERE r) = ?j :: LESS(LENGTH v). S_SEM (SEL v (0,j)) c r)
/\
(F_SEM v c (F_WEAK_SERE r) =
  !j :: LESS(LENGTH v). F_SEM (CAT(SEL v (0,j),TOP_OMEGA)) c (F_STRONG_SERE r))
/\
(F_SEM v c (F_NEXT f) =
  ?j k :: LESS(LENGTH v).
    j < k                        /\
    CLOCK_TICK (SEL v (0,j)) c   /\
    CLOCK_TICK (SEL v (j+1,k)) c /\
    F_SEM (RESTN v k) c f)
/\
(F_SEM v c (F_UNTIL(f1,f2)) =
  ?k :: LESS(LENGTH v).
    B_SEM (ELEM v k) c /\
    F_SEM (RESTN v k) c f2 /\
    !j :: LESS k. B_SEM (ELEM (COMPLEMENT v) j) c ==> F_SEM (RESTN v j) c f1)
/\
(F_SEM v c (F_ABORT (f,b)) =
  F_SEM v c f
  \/
  ?j :: LESS(LENGTH v). B_SEM (ELEM v j) b /\ F_SEM (CAT(SEL v (0,j-1),TOP_OMEGA)) c f)
/\
(F_SEM v c (F_CLOCK(f,c1)) = F_SEM v c1 f)
/\
(F_SEM v c (F_SUFFIX_IMP(r,f)) =
  !j :: LESS(LENGTH v). S_SEM (SEL (COMPLEMENT v) (0,j)) c r ==> F_SEM (RESTN v j) c f)
```

5. Definitions and proofs

The HOL versions of the semantics given in the preceding sections were not the actual definitions of the semantic functions `US_SEM`, `S_SEM`, `UF_SEM` and `F_SEM`, but were theorems derived from reformulations of the LRM definitions to make them fall within the scope of the HOL definitional tools provided by the TFL package [Sli96]. Definitions in HOL simply declare of a name for an existing closed term. Recursive 'definitions' are made by compiling equations into primitive definitions (using recursion theorems), making the definition using HOL's definition mechanism, and then deriving the equation one wants. For simple recursive equations this is handled completely automatically by TFL. For recursions that are not simple there are two options: (i) supply a proof script when making the definition (which typically involves giving some well-founded relation that ensures the recursion terminates on all arguments), or (ii) first defining a simple recursion and then deducing the desired 'definitional' equation as a theorem. We used approach (ii) for the PSL 1.1 semantics (approach (i) was used with the 1.01 semantics).

As an example, consider the definition of the unclocked semantics of the repetition SERE r`[*]` (the same issue arises with the clocked semantics). The definition of $v \models r$ is mostly by a structural recursion on the syntax of SEREs r. However, the clause defining $v \models r[*]$ does not recurse on r, but instead on v:

$$v \models r[*] \;=\; v \models [*0] \vee \exists v_1 v_2.\ \neg(v = \epsilon) \wedge (v = v_1 v_2) \wedge v_1 \models r \wedge v_2 \models r[*]$$

Observe that $v_2 \models r[*]$ occurs in the right hand side of the equation. TFL cannot automatically prove that this LRM semantics is well-founded.

The actual definition used in HOL for the r`[*]` case does recurse on r and is:

$$v \models r[*] \;=\; \exists vlist.\ (v = \mathsf{Concat}\ vlist) \wedge \mathsf{All}(\lambda v'.v' \models r)vlist$$

where $\mathsf{Concat}\ vlist$ concatenates (flattens) a list of lists and $\mathsf{All}\ P\ vlist$ applies a predicate P to each member of *vlist* and conjoins the results (i.e. combines the results with $\wedge$). The LRM equation is then deduced from the definition with Concat and All

In both the FAC and CHARME papers we described theorems about the semantics that had been mechanically proved using the HOL system. These were either 'sanity checking' properties that helped validate the semantics (FAC paper), or reformulations of the semantics needed to support tools that worked by deduction (CHARME paper).

So far we have only proved a few properties of the 1.1 semantics. These are of the 'sanity checking' kind and are taken from the first page of an unpublished paper entitled *Some characteristics of Accellera PSL* by Cindy Eisner, Dana Fisman and John Havlicek. The lemmas proved in HOL so far are all about SEREs:

$$\vdash\ \mathsf{ClockTick}(v,\mathrm{T}) \;=\; \exists kl.\ \neg(l = \bot) \wedge (v = \top^k[l])$$

$$\vdash\ \forall rvc.\ |v| > 0 \wedge \mathsf{ClockFree}(r) \wedge v \overset{c}{\models} r \Rightarrow v^{|v|-1} \Vdash c$$

$$\vdash\ \forall r.\ \mathsf{ClockFree}(r) \Rightarrow \forall v.\ v \models r[+] \;=\; \exists vlist.\ (v = \mathsf{Concat}\ vlist) \wedge |vlist| > 0 \wedge \mathsf{All}(\lambda v.v \models r)vlist$$

$$\vdash\ \forall rcv.\ v \overset{c}{\models} r[+] \;=\; \exists vlist.\ (v = \mathsf{Concat}\ vlist) \wedge |vlist| > 0 \wedge \mathsf{All}(\lambda v.v \overset{c}{\models} r)vlist$$

$$\vdash\ \forall r.\ \mathsf{ClockFree}(r) \Rightarrow \forall v.v \models r \Rightarrow \mathsf{BottomFree}(v)$$

$$\vdash\ \forall rcv.\ v \overset{c}{\models} r \Rightarrow \mathsf{BottomFree}(v)$$

$$\vdash\ \forall rv.\ \mathsf{ClockFree}(r) \wedge v \models r \Rightarrow \forall k \in [0..|v|).\ v^{0..k}\top^{(|v|-k-1)} \models r$$

In these lemmas, $\mathsf{ClockFree}(r)$ is defined to mean that r has no sub-term containing `@` (i.e. is in the unclocked subset), $\mathsf{BottomFree}(v)$ is defined to mean that no letter of v is $\bot$ and $r[+]$ is syntactic sugar for $r; r[*]$ (which in raw HOL is `S_CAT(`r`,S_REPEAT `r`)`). All these lemmas were routine to prove (though the r`[*]` case of the last lemma was surprisingly tedious).

The representation of these lemmas in raw HOL is:

```
|- CLOCK_TICK v B_TRUE = ?k l. ~(l = BOTTOM) /\ (v = TOP_ITER k <> [l])

|- !r v c.
     LENGTH v > 0 /\ S_CLOCK_FREE r /\ S_SEM v c r ==>
     B_SEM (ELEM v (LENGTH v - 1)) c

|- !r.
     S_CLOCK_FREE r ==>
     !v.
       US_SEM v (S_NON_ZERO_REPEAT r) =
       ?vlist.
         (v = CONCAT vlist) /\ LENGTH vlist > 0 /\
         ALL_EL (\v. US_SEM v r) vlist

|- !r c v.
     S_SEM v c (S_NON_ZERO_REPEAT r) =
     ?vlist.
       (v = CONCAT vlist) /\ LENGTH vlist > 0 /\
       ALL_EL (\v. S_SEM v c r) vlist
```

```
|- !r. S_CLOCK_FREE r ==> !v. US_SEM v r ==> BOTTOM_FREE v

|- !r c v. S_SEM v c r ==> BOTTOM_FREE v

|- !r v.
     S_CLOCK_FREE r /\ US_SEM v r ==>
     !k::LESS (LENGTH v).
       US_SEM (SEL v (0,k) <> TOP_ITER (LENGTH v - k - 1)) r
```

We hope to prove more properties about SEREs and also some properties about formulas. In particular, validating rewrites that translate clocked to unclocked SEREs and formulas is necessary to support our tools based on the semantics.

6. Acknowledgements

The work described here would not have been possible without the help of Cindy Eisner and Dana Fisman from the PSL/Sugar team at IBM. Dana Fisman supplied the LaTeX sources for the extracts of the draft Version 1.1 LRM shown in the framed boxes. Keith Wansbrough's HOL-to-LaTeX tool for typesetting the ASCII syntax of HOL terms was invaluable for generating the pretty-printed versions of the HOL semantics. This work was supported by an IBM Faculty Award.

References

[GHS03] Mike Gordon, Joe Hurd, and Konrad Slind. Executing the formal semantics of the Accellera Property Specification Language by mechanised theorem proving. In Daniel Geist and Enrico Tronci, editors, *Proc. 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2003)*, Lecture Notes in Computer Science. Springer-Verlag, October 2003. 21 - 24 October 2003, University of L'Aquila, Computer Science Department, L'Aquila, Italy.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem-proving environment for higher-order logic.* Cambridge University Press, 1993.

[Gor03] Michael J. C. Gordon. Validating the PSL/Sugar Semantics Using Automated Reasoning. *Formal Aspects of Computing*, 15(4):406–421, 2003.

[Sli96] K. Slind. Function definition in higher order logic. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, Turku, Finland, August 1996: Proceedings*, volume 1125 of *Lecture Notes in Computer Science*, pages 381–397. Springer-Verlag, 1996.