HEWLETT PACKARD

An Overview of HP-SL

Stephen Bear
Information Management Laboratory
HP Laboratories Bristol
HPL-91-31
March, 1991

formal;
rigorous;
specification;
industry

The Software Engineering Department of HP Labs is developing and applying a small but powerful specification language, HP-SL. This report provides an overview of the language, its supporting tools and the way in which it is being applied.

Internal Accession Date Only

1 Introduction

The Software Engineering Department of HP Labs in Bristol is working to apply, extend and transfer the use of formal methods in an industrial environment.

Two programmes of work are underway to achieve these goals. One programme is developing an *industrial* specification language, HP-SL. The other programme is establishing and supporting the use of formal specification on real product developments.

This short paper reports some of our experience. Section 2 is the main part of the report; it provides an informal overview of the specification language HP-SL. Section 3 describes the tools developed to support HP-SL. Section 4 sketches our approach to transferring formal specification into HP.

1.1 Specification for Industry

Industrial training in formal specification must address two issues simultaneously.

- The idea of specifying a system by giving mathematical definitions.
- The details of a language in which to express such definitions.

Training at the start of a project is a visible cost, so it is important to be able to present these ideas quickly and efficiently. An industrial specification language must reduce the 'language overhead' as much as possible. However, the language must not be overly restricted. It must be flexible enough and powerful enough to define varied products in a natural way. In particular the language should not impose a single idiom.

HP-SL has been designed to address these concerns. It is a small and regular language. Two concepts are fundamental: abstract types and underspecified total functions. Building upon this foundation, the language provides the usual specification concepts. It is also flexible. Definitions may be given in assertional, pre-post or explicit styles. These different approaches are integrated in a uniform framework, so a single function may be specified in any mixture of the styles.

Polymorphic functions and 'new' type constructors may be defined. This makes it possible to effectively extend the language and to create new idioms. A very successful example, *'History Specification'* is described in [3].

A complete definition of HP-SL is beyond the scope of this paper, but a more detailed description of the language may be found in [5]

2 An Overview of HP-SL

The objective of this section is to give an overview of the HP-SL language and the specification style that it supports. We assume that the reader is familiar with other model oriented specification languages, such as VDM [1].

An HP-SL specification consists of a collection of definitions—definitions of types, values of types, functions or relations over the types, and assertions about types or values. The definitions may be interspersed with narrative text.

2.1 Basics

Predefined Types

HP-SL provides a number of pre-defined types including *Bool*, *Char* and *Real*. Values of these types may be defined explicitly or implicitly.

```
val ten : Real ≜ 10
val delta : Real sat delta > 0 ∧ delta ≤ 0.05
val number : Real
```

These definitions introduce three named values: *ten* is an explicitly defined value of type *Real*; *delta* is an underspecified value of type *Real* which satisfies the constraint $delta > 0 \wedge delta \leq 0.0$ *number* is an underspecified value of type *Real* which is not constrained.

Predefined Type Constructors

HP-SL provides a number of pre-defined type constructors. These include the set and sequence type constructors *Set*, *Seq*, the map and function type constructors $\xrightarrow{m}$, $\rightarrow$ and the tuple type constructor $\times$. Later, we will see how new type constructors may be defined.

```
val realset : Set Real ≜ { 1, 2, 3, 4 }
val realseq : Seq Real ≜ ≪ 1, 2, 3, 4 ≫
val realmap : Real ⇸ Real ≜ [ 1 ↦ 1, 2 ↦ 4, 3 ↦ 9 ]
val realfun : Real → Real ≜ ( λ x : Real · x*x )
val realpair : Real × Real ≜ (1, 2)
```

This fragment of HP-SL defines values of a number of types: *realset* is the set of real numbers 1,2,3 and 4. *realseq* is the sequence of real numbers 1,2,3,4. *realmap* is the map of the real numbers 1,2,3 to their squares. *realfun* is the function of the real numbers to their squares. Finally *realpair* is the pair of real numbers (1, 2).

Synonym Types

It is often convenient to give a name to an existing type or a type expression. In HP-SL this is done by a synonym type definition.

$$\textbf{syntype}\ \mathit{RealPair} \triangleq \mathit{Real} \times \mathit{Real}$$

This definition introduces a new name *RealPair* for the type denoted by the expression *Real* $\times$ *Real*. The definition does not introduce a distinct type. A value of type *RealPair* may be used wherever a value of type *Real* $\times$ *Real* could be used, and vice-versa.

Subtypes

Sometimes it is useful to define those elements of a type which satisfy some property. Such a collection is called a *subtype*. The property satisfied by elements of the subtype is called the invariant.

In HP-SL a subtype is defined by a type expression with an invariant clause. A subtype is often named by a synonym type definition.

$$\textbf{syntype}\ \mathit{Positive} \triangleq \mathit{Real}\ \textbf{inv}\ r \cdot r > 0$$

This fragment of HP-SL defines a subtype of type *Real* which consists of all real numbers which are strictly greater than zero. The subtype is defined by the type expression *Real* **inv** $r \cdot r > 0$. The invariant clause is introduced by the language word **inv** which is followed by a pattern r and then the invariant predicate $r > 0$.

HP-SL provides a number of predefined subtypes, including *Int*, *Nat0* and *Nat1*.

2.2 Functions

Functions may be defined using a typed lamda calculus.

```
val realsqr : Real → Real ≜ ( λ x : Real · x*x )
```

realsqr is a function of type *Real* → *Real* defined by the lambda expression $\lambda\ x : Real \cdot x*x$.

As a derived form, HP-SL provides a syntax which is closer to that used in programming languages, and is more familiar to engineers.

```
fn realsqr : Real → Real is
   realsqr( x ) ≜ x*x
```

These two definitions of *realsqr* are equivalent.

Strictly, HP-SL functions have precisely one argument—multiple arguments are combined by the tuple type constructor. However, function application is by juxtaposition, so the resulting syntax is simple and looks natural.

```
fn realmultiply : Real × Real → Real is
   realmultiply( x, y ) ≜ x*y

val six ≜ realmultiply( 2, 3 )
```

HP-SL functions are higher-order. Functions may accept other functions as arguments. In particular, functions may be defined in a 'curried' form.

```
fn realmultiply : Real → Real → Real is
   realmultiply( x )( y ) ≜ x*y
```

Curried functions may be partially applied.

```
val double: Real → Real ≜ realmultiply(2)
```

double is a function of one real which doubles its argument.

Implicit Definitions

Implicit definitions are frequently more convenient than explicit definitions. HP-SL implicit definitions are conventional—a post-condition is given to define the relationship between the value returned by a function and the values of its arguments.

```
fn prime_factors : Natl → Seq Natl is
    prime_factors(n)
    return s
  post
    product(s) = n ∧
    ( ∀ p ∈ elems(s) · is_prime(p) )
```

This is equivalent to defining a value of the function type which satisfies the post condition.

```
val prime_factors : Natl → Seq Natl sat
  ( ∀ n : Natl ·
    let val s ≜ prime_factors(n) in
      product(s) = n ∧
      ( ∀ p ∈ elems(s) · is_prime(p) )
    endlet
  )
```

Partial Functions

Functions are not always well-defined for every value in the domain type, for example standard real division is not well-defined for divisor 0.

In HP-SL all functions are *total* functions, but a pre-condition may be used to explicitly

indicate where a definition applies. For example, a *divide* function could be defined as follows

```
fn divide : Real → Real → Real is
    divide(divisor)(numerator)
  pre divisor ≠ 0
    return quotient
  post quotient * divisor = numerator
```

The pre-condition in this definition does not restrict the *domain* of the function, it indicates where the definition constrains the function. It is equivalent to

```
fn divide : Real → Real → Real is
    divide(divisor)(numerator)
    return quotient
  post
    divisor ≠ 0 ⇒ quotient * divisor = numerator
```

The application of a function to a value which does not satisfy the pre-condition is a valid expression. The following definition

```
val divide_by_zero : Real → Real ≙ divide 0
```

is valid. The function *divide_by_zero* is some function of type *Real* → *Real* but it is not constrained by the above definition of *divide*.

2.3 Incremental Definitions

When dealing with large systems it is convenient to be able to present specifications 'bit by bit'. In HP-SL, values may be defined by a series of definitions. Consider the following trivial example

```
val zero : Real
assert non_positive ≜ zero ≤ 0
assert non_negative ≜ zero ≥ 0
```

This fragment of HP-SL consists of three definitions. The first definition introduces *zero* an unconstrained value of type *Real*.

The next definition is an 'assertion' called *non_positive*. An assertion is not a statement of some property which can be proved from other definitions. It is a definition which imposes a constraint on the specification. In this case it constrains the value *zero* to be less than or equal to 0. The second assertion, *non_negative* further constrains the value to be greater than or equal to 0. The overall effect is to define a value which satisfies both constraints.

Above we gave a definition of the function *divide* which constrained the function for non-zero divisors. We can give an assertion which ensures that division by zero always returns the value 0.

```
assert zero_divisors ≜ ( ∀ x:Real · divide(0)(x) = 0 )
```

An alternative approach is to give *multiple* definitions. The following further definition of *divide* constrains the function for zero divisors. It has the same effect as the assertion—it ensures that division by zero returns the value 0.

```
fn divide : Real → Real → Real is
   divide (divisor)(numerator)
      pre divisor = 0
   ≜ 0
```

2.4 Abstract Types

Defining New Types

In HP-SL, a 'new' type is introduced by an type definition. This should be contrasted with a synonym type definition which just introduces a new name for an existing type.

```
type Person
type Contents
```

The type *Person* and the type *Contents* are new types which are distinct from all other types.

Such types are called *abstract types* because they have no 'internal structure', other than that implied by functions operating on the type. If there are no functions which operate on *Person* and *Contents* we know nothing more about them. This is equivalent to stating that properties and attributes of the type are irrelevant to the specification.

Abstract Types and Explicit Assertions

We can impose some structure on an abstract type by defining functions, and explicitly constraining the functions by assertions. We will explain this by presenting an example in detail. Afterwards, we will discuss some shorthand which makes such definitions much easier to write.

```
type Message

fn message : Person × Set Person × Contents → Message
fn sender : Message → Person
fn recipients : Message → Set Person
fn contents : Message → Contents
```

These definitions introduce a new abstract type, called *Message*, and four unconstrained functions: one which constructs values of type *Message* and three which 'project' into other types. We can constrain these functions by giving an assertion.

```
assert message_projectors ≜
  ( ∀ ( p:Person, sp:Set Person, c:Contents ) ·
      sender( message(p,sp,c) ) = p ∧
      recipients( message(p,sp,c) ) = sp ∧
      contents( message(p,sp,c) ) = c )
```

The assertion ensures that, for any value of type *Message* constructed by the *message* function, the projector functions return the appropriate components.

A consequence of this assertion is that there is no 'confusion' between values constructed by *message*. Two values $message(p_1,sp_1,c_1)$ and $message(p_2,sp_2,c_2)$ of type *Message* are equal if and only if $p_1 = p_2$ and $sp_1 = sp_2$ and $c_1 = c_2$.

We may also want to say that *all* values of type *Message* can be constructed by the constructor function. This is ensured by the following definition.

```
assert no_junk_message ≜
  ( ∀ m:Message · ( ∃ ( p:Person, sp:Set Person, c:Contents ) ·
      message(p,sp,c) = m ) )
```

This example shows that an abstract type may be defined by giving underspecified functions and then constraining the functions by assertions. This is a powerful approach, but it is too complex for routine industrial use. In the next section we will look at some 'shorthand' derived syntax which provides easy ways to give common definitions.

Record Types

The abstract type *Message* was defined by giving a constructor function, projectors, and explicit assertions which ensured that the type was isomorphic to $Person \times Set\ Person \times Contents$. One way to think of such a definition is as a 'tagged record' type. This is a very common kind of definition, and HP-SL provides a convenient short syntax.

```
type Message ≜
  [ message ▷
      ( sender : Person,
        recipients : Set Person,
        contents : Contents )   ]
```

This definition of the type *Message* is equivalent to that given in the previous section. The constructor function, projector functions and the assertions are derived systematically from the syntax.

The details of the type are contained within special brackets **[** ... **]**. The name of the constructor function *message* is followed by a delimiter ▷. This is followed by the names and

type of the 'components'. These names allow us to derive the signatures of the constructor and projector functions.

```
fn message : Person × Set Person × Contents → Message
fn sender : Message → Person
fn recipients : Message → Set Person
fn contents : Message → Contents
```

A number of other functions are also derived. These include an *'is'* function.

```
fn is_message : Message → Bool is
    is_message(m) ≜
      (∃ (p:Person, sp:Set Person, c:Contents) · message(p,sp,c) = m )
```

The *projectors* assertion and the *no junk* assertion may also be derived systematically. The *no junk* assertion may be stated in terms of the *is* function.

```
assert no_junk_message ≜
  ( ∀ m:Message · is_message(m) )
```

So the 'record type' syntax is a shorthand for the full definition of the abstract type, the constructor and projectors, and the assertions. We will call the derived functions and assertions, 'default' functions and assertions. They may be used just as if they had been defined explicitly.

Multiple Constructors

It is straightforward to extend this approach to allow more than one constructor. Syntactically, constructors are just separated by a vertical bar |.

```
type RealTree ≜
  [ leaf ▷ leaf_value : Real ] |
  [ node ▷
      ( left : RealTree,
        right : RealTree ) ]
```

This definition defines a type of tree where values are stored at the *leaves.*

A number of default functions and assertions are derived from the definition. Firstly, there are *two* constructors

```
fn leaf : Real → RealTree
fn node : RealTree × RealTree → RealTree
```

and their associated projectors.

```
fn leaf_value : RealTree → Real
fn left : RealTree → RealTree
fn right : RealTree → RealTree
```

The default *projector* and *no junk* assertions are derived. The *projector* assertions apply to each constructor separately. The *no junk* assertion applies jointly—it ensures that all values of type *RealTree* are constructed by either *leaf* or *node*. It may be stated using the default *is* functions.

```
assert no_junk_realtree ≜ ( ∀ t : RealTree · is_leaf(t) ∨ is_node(t) )
```

Two further default assertions are derived: *no confusion* and *induction.*

If a type has multiple constructors, the *projector* assertions are not enough to ensure that there is no confusion. We need to ensure that the values constructed by different constructors are distinct. This is achieved by the *no confusion* assertion.

```
assert no_confusion_realtree ≜
  ( ∀ t:RealTree · ¬ ( is_leaf(t) ∧ is_node(t) ))
```

Together with the projector assertions, this ensures that values of *RealTree* are equal if and only if they are constructed by the same constructor *and* the corresponding constructor arguments—components—are equal.

If a type is 'recursive'—that is, if a constructor function includes the type in its domain—then the *no junk* assertion is not enough to ensure that we can reason about *all* values of the type. This is achieved by the *induction* assertion.

```
assert induction_realtree ≜
  ( ∀ p : RealTree → Bool ·
     (∀ x:Real · p(leaf(x)) ) ∧
     (∀ (t1:RealTree, t2:RealTree) · p(t1) ∧ p(t2) ⇒ p(node(t1,t2)))
     ⇒
     (∀ t:RealTree · p(t))
   )
```

Constant constructors

Often, we wish to give constants as well as constructor functions. This is done by allowing the constructors to be named constants.

```
type RealStack ≜
  [ empty ] |
  [ push ▷
      ( pop : RealStack,
        top : Real )     ]
```

The type *RealStack* has two constructors: the constant *empty* and the function *push* : $RealStack \times Real \rightarrow RealStack$.

The constant *empty* does not have any projectors. The constructor *push* has two projectors:

pop and *top*.

Together with the default derived assertions, this is essentially equivalent to the usual initial algebra definition of a stack abstract datatype.

Enumerated Types

Sometimes, we want to define a type by enumerating its values. In HP-SL we simply give the constructor constants.

```
type Colour ≙ [ red ] | [ green ] | [ blue ]
```

The type *Colour* consists of precisely the constants *red, green* and *blue*. The default no junk assertion ensures that there are no other values, and the default no confusion assertion ensures that they are distinct values.

2.5 Relations

Specifying relationships between values is a fundamental specification technique. The basic approach is to define relationships by Boolean valued functions. For example, consider

```
fn less_than : Int × Int → Bool is
    less_than(x,y) ≙ x < y
```

The values x and y are related, if and only if the expression $less\_than(x,y)$ is *true*.

In model oriented specification, such definitions are very common. HP-SL provides a derived syntax which emphasises the fact that we are interested in the *relationship* rather than the value returned by the function. For example the function *less_than* may be written as follows.

```
reln less_than : Int × Int is
      less_than(x,y) ≙ x < y
```

2.6 State and Operations

A typical model oriented specification of a system provides definitions of system state and operations which can update that state.

In HP-SL the system state is modelled by a type; operations on the system state are modelled by functions or relations.

HP-SL does not distinguish the system state type—it is defined and referenced in exactly the same way as other types. Its special role in the specification is explained by the narrative text and not by the syntax. One consequence of this approach is that there is no 'frame-condition'—if an operation leaves part of the state unchanged, then the definition must say so.

To illustrate this style we give a specification of a trivial spell-checker system. The system maintains a dictionary of known words and has two operations. The first checks whether or not a given word is in the dictionary; the second will add a given word to the dictionary.

The dictionary contains words. We do not need to know anything about the properties of words, so they are modelled by an abstract type, *Word*.

type *Word*

The system dictionary is a collection of words, modelled as a set of words.

syntype *Dictionary* $\triangleq$ *Set Word*

The *checkword* operation takes a single word and checks whether or not it is in the dictionary. If it is in the dictionary, it returns the value *true*; if not, it returns the value *false*. In either case, the system dictionary is not changed. We will model this operation as a function.

fn *checkword* : *Word* $\times$ *Dictionary* $\rightarrow$ *Bool* **is**
checkword(*word*, *sys_dictionary*) $\triangleq$ *word* $\in$ *sys_dictionary*

In this definition, the word to be checked is *word* and the system dictionary is *sys_dictionary*. Since this is a function, the system dictionary is not changed.

The *addword* operation takes a single word and adds it to the dictionary. We model this operation by a relation.

```
reln addword : Dictionary × Word × Dictionary is
     addword( sys_dictionary, word, sys_dictionary/ )
     ≙ sys_dictionary/ = sys_dictionary ∪ {word}
```

In this definition, the initial value of the system dictionary is *sys_dictionary*, the word to be added is *word* and the final value of the system dictionary is *sys_dictionary/*. The choice of names for the formal parameters of the relation are, of course, quite arbitrary. Current use of HP-SL follows the convention that the name of the initial value of system state is undecorated, and the final value is decorated by a prime.

2.7 Polymorphism

Polymorphic Functions

In HP-SL, it is possible to give *polymorphic* definitions. For example, the following defines a polymorphic function which returns the *last* element of a sequence. (The operator *elem* turns the sequence into a function).

```
fn (| T |) last_element : Seq T → T is
   last_element( s )
     pre len s ≠ 0
     ≙ (elem s)(len s)
```

The special brackets (| |) at the start of the definition introduce a *type variable*, T, which represents *any* type.

A polymorphic definition may be thought of as a finite representation of an infinite family of definitions—one for each binding of the type variable to a type. The above polymorphic definition introduces, amongst others, the functions

```
fn last_element : Seq Real → Real
fn last_element : Seq Bool → Bool
```

```
fn last_element : Seq (Real × Real) → Real × Real
...
```

Type Constructors

Polymorphism also allows us to give new *type constructors.* For example we can define a general *Tree* type constructor.

```
type (| T |) Tree ≜
        [[ leaf ▷ leaf_value : T ]] |
        [[ node ▷
            ( left : Tree T,
              right : Tree T ) ]]
```

This introduces an abstract type constructor, and associated polymorphic default functions and assertions.

A new type constructor may be used in exactly the same way as a pre-defined type constructor. In the following example, notice that there is no distinction between the pre-defined constructor *Set* and the new type constructor *Tree*.

```
fn (| T |) leafvalues : Tree T → Set T is
   leafvalues( tree ) ≜
     if is_leaf(tree) then {leaf_value(tree)}
     else leafvalues( left(tree) ) ∪ leafvalues( right(tree) )
     endif
```

The function *leafvalues* collects the set of values associated with the leaves of a tree.

We can also define *synonym type* constructors,

```
syntype (| T |) Bag ≜ T →m Natl
```

and give associated functions explicitly.

```
fn (| T |) count_element : T × Bag T → Nat0 is
    count_element ( elem, bag ) ≜
      if elem ∈ dom bag then lookup bag(elem) else 0 endif
```

3 Tools to support HP-SL

The HP-SL toolset supports a style of *literate specification* by allowing the production of documents containing both formal specification and narrative text. Specifications written in HP-SL can be easily incorporated into documents written using the LaTeX document preparation system.

The toolset provides syntax and incremental typechecking, via an interface built upon the GNU Emacs editor [4]. A document is prepared in, or read into, a normal Emacs buffer. Commands bound to Emacs key-sequences invoke the parser and, optionally, the type checker. The parser and type checker work upon an item or region, which contain one or more HP-SL definitions. Definitions may be parsed in isolation; the type checker maintains an 'environment' of type information.

Incremental checking provides fast response and encourages developers to check their work as they write it. Any errors message from the parser or type checker are displayed in a separate Emacs window. Commands are available to move the editor cursor to the point at which each error is detected.

A specification document is formatted by a filter which replaces HP-SL Ascii syntax with appropriate LaTeX commands, and then invokes the normal LaTeX to DVI translator. The resulting document may be printed, or previewed at a workstation.

The toolset also provides a network-transparent database which allows several different documents, perhaps written by several members of a project, to share the same pieces of HP-SL specification. Items in the database are under version control and a document may refer to a particular version of an item, or to the 'latest' version, simplifying the task of keeping several separate but related documents in step during the design process.

4 Technology Transfer

A major objective of the Software Engineering Department is to demonstrate that the appropriate use of formal specification results in faster development of better quality software. We are working with a series of projects which can be used internally as examples and case studies.

Our current partners include two medical products, a CAD system and a pure software system. These are real product developments, not investigations or feasibility studies.

We have established a technology transfer model which helps to ensure that, even in the short term, the process of learning and using the technology is a net benefit.

The most important aspect is that we provide *project centred* training and support. We do not give general courses in the hope that some people might try out the ideas. We establish a close contact with a project team and work with them throughout the product development.

The process begins by identifying a suitable project. The criteria which we apply include the following.

- The project should be a mainstream product development involving new software.
- The project should be scoped, but work on the software must be at an early stage.
- The software team should be reasonably small, and enthusiastic about using a better development approach

Once a project has been identified, there is an investigation phase, which we use to develop an understanding of the application area. As a result of the investigation a joint proposal is made; this explains how formal specifications will be applied to the project and what benefits are expected. It is important that the management chain supports the collaboration.

The collaboration is launched with a two week visit by two people from HP Labs. The first week is a training course which covers simple discrete maths and the HP-SL language. The last day of the course is used to develop a prepared case study. The second week is a workshop which begins the process of applying HP-SL to the project itself. By the end of the workshop there is a firm technical strategy, and the project is using the specification language independently.

After the project launch we stay in contact with the project. We provide support by regular use of telephone, electronic mail and teleconference links. Further visits, by both sides, take place as the project proceeds.

All of our collaborations are still in progress, and none has yet reached product release, but initial results have been very encouraging. The language and the approach are transferable and help provide a faster, more efficient development process.

5 Acknowledgements

HP-SL is a model oriented specification language in the tradition of VDM, [1] and RAISE [2]. The concept and the detailed design of HP-SL are due to Patrick Goldsack. Errors and

omissions in this presentation are mine.

6 References

[1] Jones C B. *Systematic Software Development Using VDM.* Prentice-Hall, Second edition, 1990.

[2] Havelund K. and Haxthausen A. RSL Reference Manual. Technical Report RAISE/CRI/DOC/2/V1, Computer Resources International, 1990.

[3] Harry P. History Specifications. HPL Technical Memo in Preparation, 1991.

[4] Stallman R. The extensible, customizable, self documenting, display editor. In *Interactive Programming Environments.* McGraw-Hill, 1984.

[5] Rush T., Harry P., Ferguson T., and Oliver H. Case studies in HP-SL. Technical Report HPL-90-137, Hewlett-Packard Laboratories, Bristol, 1990.