

A Universal Intermediate Representation for Massively Parallel Software Development

Paul Damian Wells
23025 NE Mountaintop Rd.
Newberg, OR. 97132
pdamianw@acm.org

Abstract: The Hierarchical Simultaneous Set Membership (HSSM) intermediate representation is a graphical representation that can be used to efficiently encode source languages during parsing. It also provides an excellent design representation for code generation as well. Moreover, the HSSM IR is the basis for a *relational algebra*. Where a relational algebra is to software design what Boolean algebra is to digital logic design. This formal mathematical interpretation makes the HSSM IR an excellent platform on which to develop the extensive automated synthesis/verification algorithms that will necessarily form the basis of next generation computers.

Keywords: Intermediate Representation; Relational Algebra; Device Oriented Programming (DOP)

Introduction

Is there any distinction between "programming" and "software engineering"?

Yes there is! "Software Engineering", like all engineering disciplines, is applied mathematics - the application of formal mathematical models to the solution of real world problems. "Programming" is just a skill-based profession. An ad-hoc trial-and-error approach to software development.

If you are only interested in relatively small, non-parallel software, you can certainly use an ad-hoc style and be very successful. If on the other hand, you want to design the large, massively parallel software of the future, the formal methodology of software engineering must be adopted. Simply trying to "program" software projects of this size would be analogous to building a skyscraper with a hammer and nails.

This paper is about software engineering. In particular, it presents a formal mathematical model specifically developed for use in software design: The HSSM Intermediate Representation (IR). The HSSM IR is the graphical representation of a relational (predicate) algebra. The representation of software using this formal mathematical model provides the foundation on which to build newer and advanced Computer Aided Design (CAD) tools. These tools in turn will provide a software design platform sophisticated enough to support massively parallel software development.

Overview

A software design in the HSSM IR is represented as a collection of rules. These rules come in two forms; a *conditional disjunction statement* and a *conjunctive collection statement*.

$$A \leftrightarrow [B ? C \mid D].$$

$$B \leftrightarrow \{ E, F, G \}.$$

As shown, each type of rule consists of a *conclusion* to the left of the equivalence operator and a *hypothesis* to the right. The hypothesis of a conditional disjunction statement is the triadic conditional disjunction operator. The hypothesis of a conjunctive collection is one or more terms in curly braces. These two types of rule may be rewritten using the more familiar logical operators conjunction, negation and disjunction as shown below:

$$A \leftrightarrow (B \wedge C) \vee (\neg B \wedge D).$$

$$B \leftrightarrow (E \wedge F \wedge G).$$

When a term within the hypothesis of a conditional disjunction statement matches the conclusion of a conjunctive collection statement, the term may be replaced by the hypothesis of the conjunctive collection. The two rules above for example, may be combined into a single rule as shown below. Any software design may be represented in the HSSM IR as a collection (database) of these types of rule.

$$A \leftrightarrow [\{ E, F, G \} ? C \mid D].$$

In reviewing the HSSM IR, be aware that the HSSM IR is an "algebra" not a "calculus". The HSSM IR is to software design what Boolean Algebra is to digital logic design. Where the Boolean Algebra is a distilled version of the zero order predicate calculus, the HSSM IR is a distilled version of the first order predicate calculus. A calculus is primarily the tool of a mathematician, where an algebra is primarily the tool of an engineer.

The next two pages present a derivation of the HSSM IR from simple set theory. The following six pages are then used to present the HSSM Intermediate Language (HIL). HIL is a programming language derived from the HSSM IR and serves to illustrate the utility of the IR for software development. The final page contains a summary of the entire paper.

The HSSM IR

The HSSM intermediate representation is based on set theory and the universe of discourse (carrier) is the inductively defined set of tuples "T";

$$T = \{ \mathbf{x} \mid \mathbf{x} \in T^n \}$$

where $(T^n = T \times \dots \times T)$ are the cross products on the set T. This set of tuples is the union of an infinite number of infinite sets.

$$\begin{aligned} T^0 &= \{ \langle \rangle \} \\ T^1 &= \{ \langle \rangle, \langle \langle \rangle \rangle, \langle \langle \langle \rangle \rangle \rangle, \dots \} \\ T^2 &= \{ \langle \langle \rangle, \langle \rangle \rangle, \langle \langle \rangle, \langle \langle \rangle \rangle \rangle, \dots \} \end{aligned}$$

HSSM reasoning amounts to defining and proving relations over the set T. The most basic definition is a declarative statement called a *fact*. The graph on the left of Figure 2 below represents the factual declaration " $\langle \rangle \in f$ ". The empty tuple is an element of the relation "f". The graph on the right of Figure 2 represents the declaration " $\langle \langle \rangle, \langle \rangle \rangle \in g$ ". Equivalently, this can be written as the string "g($\langle \rangle, \langle \rangle$)."

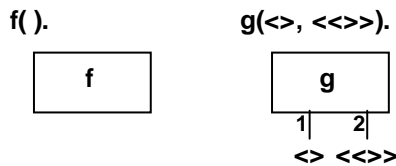


Figure 2 Factual Declarations

A fact may be extended to form a conditional disjunction statement called a *rule*. Figure 3 below, represents the statement " $\langle X, Y \rangle \in f$ if and only if either $\langle X \rangle \in g$ and $\langle X \rangle \in n$ or $\langle X \rangle \notin g$ and $\langle Y \rangle \in a$ ". The fact forms the *conclusion* of the rule while the *hypothesis* is composed of three ordered *queries* called the *guard*, *normal* and *abnormal* respectively.

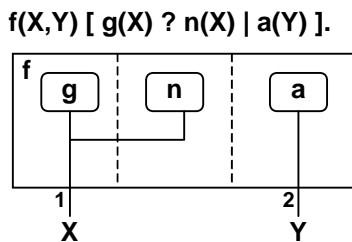


Figure 3 Conditional Disjunction Statement (rule)

The queries of the hypothesis are drawn as rounded squares inside the body of the fact in order left to right. As shown, *variables* may be used to represent tuples. Variables are quantified across the entire rule and bound variables are drawn connected into a single net. A rule may be expressed using the equivalent string notation: "f(X,Y) [g(X) ? n(X) | a(Y)]." The guard is used to select either the normal or abnormal query and the fact is true if and only if the selected query is true.

Conjunctive Collections

A *collection* is a set of zero or more rules that collectively defines a relation. The ternary set of rules shown just below is a *conjunctive collection*. The abnormal query in each rule of a conjunctive collection always evaluates to false (contradiction) and the rules form a chain.

$$\begin{aligned} \{ & f(X,Y) [a(X) ? g(X,Y) \mid T^1(\cdot)]. \\ & g(X,Y) [b(Y) ? h(X,Y) \mid T^1(\cdot)]. \\ & h(X,Y) [c(X,Y) ? T^0(\cdot) \mid T^1(\cdot)]. \} \end{aligned}$$

Conjunctive collections occur so frequently; they may be expressed with a shorthand rule notation. The *conjunctive collection statement* shown just below is equivalent to the conjunctive collection above.

$$f(X,Y) \{ a(X), b(Y), c(X,Y) \}.$$

Like a rule, a fact forms the conclusion of a conjunctive collection. The hypothesis however, is a set of queries enclosed in braces as shown. The fact is true if and only if each query of the hypothesis is true. The queries of a conjunctive collection are drawn inside the body of the fact as shown in Figure 4 below.

$$f(X,Y) \{ a(X), b(Y), c(X,Y) \}.$$

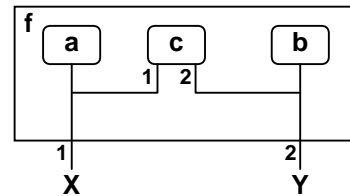


Figure 4 Conjunctive Collection

When a query corresponds to a conjunctive collection, the query may be replaced by the hypothesis of the conjunctive collection. As shown just below in Figure 5, the queries of an *inline conjunctive collection* are drawn inside the body of the fact.

$$\begin{aligned} \{ & f(X,Y,Z) [T^1(X) ? g(X,Y,Z) \mid \text{decr}(Y,Z)]. \\ & g(X,Y,Z) \{ \text{head}(X,V), \text{incr}(Y,W), f(V,W,Z) \}. \} \end{aligned}$$

$$\begin{aligned} f(X,Y,Z) [& T^1(X) \\ & ? \{ \text{head}(X,V), \text{incr}(Y,W), f(V,W,Z) \} \\ & \mid \text{decr}(Y,Z)]. \end{aligned}$$

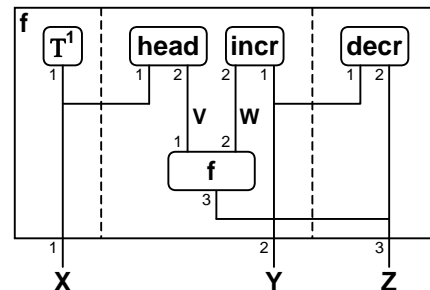


Figure 5 Inline Conjunctive Collection

Instantiation Attributes - Partial Ordering

Any variable appearing in a rule may be augmented with an *instantiation attribute* (instatt). An instatt appears as a carrot character (^) immediately preceding the variable name. An instatt declares that the variable will be instantiated during the successful evaluation of the query or rule. A variable may be instantiated only once during the evaluation of a rule; therefore the instatt imposes a partial ordering on the evaluation order of queries in a rule.

As shown in Figure 6 below, a reverse notation is used when drawing rules that contain instatts. Variables that are instantiated during a rule or query are drawn exiting a node from the bottom. Variables without an instatt are shown entering a node from the top with an arrowhead at the intersection of the net and node. The result is a dependence graph.

$f(X, Y, ^Z) [T^1(X)$
 $? \{ \text{head}(X, ^V), \text{incr}(Y, ^W), f(V, W, ^Z) \}$
 $| \text{decr}(Y, ^Z)]$.

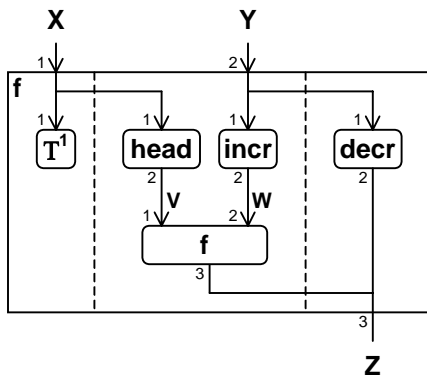


Figure 6 Instantiation Attributes

Any variable appearing in a rule without an instatt may be augmented with an *un-instantiation attribute* (unstatt). An unstatt appears as a splat character (*) immediately preceding the variable name. An unstatt declares that a variable will be un-instantiated during the successful evaluation of the rule or query. It follows that the variable must be instantiated before the query may be evaluated. Like instatts, uninstatts impose a partial ordering on the evaluation of queries in a rule.

Finally, any variable may be augmented with a *range attribute*. A range attribute appears as a colon character ':' and relation name immediately following a variable name. A range attribute declares that the variable may only be instantiated with tuples that are a member of the specified relation. For example; if a relation "bit" is defined to contain all tuples except the empty tuple as shown below, then a relation "nibble" may be defined as a set of 4-tuples such that each member of the 4-tuple is a member of the "bit" relation.

$\text{bit}(A)$.
 $\text{nibble}(W:\text{bit}, X:\text{bit}, Y:\text{bit}, Z:\text{bit})$.

HSSM Basis Relations

HSSM graphs are an inductively defined set where rules represent composition relations. A complete representation needs the addition of three basis relations: "select()", "modify()" and "equal()".

The "select()" relation contains 3-tuples used to access an element of a tuple:

$\text{select} = \{ \langle \langle X_0, \dots, X_n \rangle, m, X_m \rangle \mid m \leq n \}$

The "modify()" relation contains 4-tuples used in a similar fashion to create a copy of a tuple with a single element modified:

$\text{modify} = \{ \langle \langle X_0, \dots, X_n \rangle, m, Y, Z_0, \dots, Z_n \rangle \mid m \leq n \text{ and } Z_m = Y \text{ and } Z_{n \neq m} = X_n \}$

The "equal()" relation is a two-tuple used to create an exact copy of a tuple or check for equivalence:

$\text{equal} = \{ \langle X, Y \rangle \mid X = Y \}$

HSSM Computability

It should be noted that HSSM rules are based on the triadic *conditional disjunction* operator. This operator is a *pseudo-Sheffer function* and it forms a complete set of logical connectives by itself - if the truth-values "TRUE" and "FALSE" are represented. These values are represented in the HSSM IR as shown below:

$\text{TRUE} = T^0()$
 $\text{FALSE} = T^1()$

Finally, note that each rule in the HSSM IR is a *wff* of the predicate calculus and the HSSM IR is actually a predicate logic language. It should be obvious by inspection that HSSM rules may be mechanically converted to a *clausal form* and automated reasoning accomplished with the *resolution inference rule*.

The HSSM Intermediate Language

The preceding 17 paragraphs present the entire HSSM IR. The remainder of this paper will be used to present the HSSM Intermediate Language (HIL). The HIL is a general-purpose, intermediate-level language used to define software *devices* as a set of HIR rules. These rules may then be manipulated automatically to accomplish large-scale verification and synthesis tasks.

HIL is a *data abstraction* language that provides many of the language constructs of Object Oriented Programming (OOP). Abstract data tuples (ADTs) may be defined with *associated* rules in a manner very similar to classes and member functions in OOP.

HIL is distinguished from object oriented and imperative languages not by any new additions, but by the omission of illogical constructs. Operations such as the "goto statement", "type casts" and "inheritance with exceptions" are not supported. Yet the language is still very general purpose and useful. HIL also supports a logically robust form of parallel programming.

HIL Associated Rules

An *associated* rule in HIL is analogous to a member function in OOP. Shown just below is a typical rule definition which implements 16-bit integer division. The rule is named "divide" and it is associated with an ADT named "int16". It takes two 16-bit (field) input parameters; "X" and "Y". It returns a single 16-bit value "Z". There are two 16-bit local variables declared on the second line; "A" and "B".

```
rule int16.divide(X:field, Y:field, ^Z:field)
  <A:field, B:field>
  ? (X > Y)
    field.subtract(X, Y, ^A),
    int16.divide(A, Y, ^B),
    field.add(B, 1 ^Z)
  | field.equal(0, ^Z).
```

The third through seventh line above represent the body of the rule and may be interpreted as an if-then-else statement (conditional disjunction). The line preceded by a question mark is an infix expression that checks whether the value of "X" is greater than "Y". If it is, the value of "Y" is subtracted from "X" and the rule "int16.divide" is called recursively. Else the value zero is returned. An HSSM graph of this rule is presented in Figure 7 below.

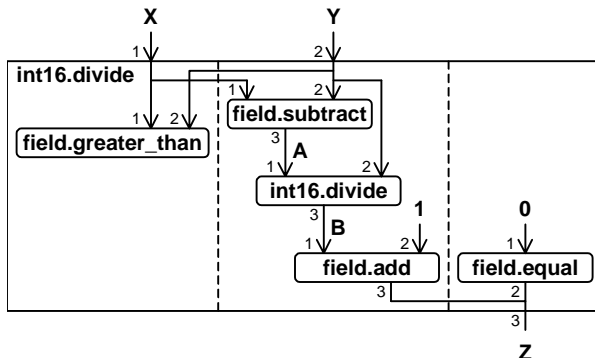


Figure 7 HSSM Graph of "int16.divide" rule

As shown, the expression preceded by the question mark is a query of the guard. The lines preceded by the vertical bar are all queries of the abnormal collection and the intervening lines are all queries of the normal collection.

Exception Handling

Like all integer division algorithms, the "int16" rule defined above is susceptible to a divide-by-zero error. If the value of the input parameter "Y" is zero, an infinite recursion will result. Error conditions like this need to be detected and handled so that rule evaluation always terminates gracefully with a truth-value of "TRUE". With the exception of infix expressions such as those used in the guard, a query should never evaluate to "FALSE".

In the first version of HIL, error handling required the creation of subordinate rules that collectively provided error checking and propagation. The result was a proliferation of rules and a pollution of the rule name space. As a result, HIL now includes built-in error checking/propagation constructs. Shown just below is the "int16.divide" rule rewritten with a distinguished error parameter named "#", a *common query* and two *dynamic range restrictions* (DRRs).

```
rule int16.divide(X:field, Y:field, ^Z:field, ^#:field)
  <A:field, B:field, E:field>
  ? (X > Y)
    field.subtract(X, Y, ^A),
    int16.divide(A, Y, ^B, ^E),
    field.add(B, 1, ^Z)
  | field.equal(0, ^Z),
  + field.equal(0, ^#)
  - (Y = 0)
    field.equal(1, ^#),
    field.equal(0, ^Z)
  - (E <> 0)
    field.equal(E, ^#),
    field.equal(0, ^Z).
```

Each DRR is marked with a minus sign and begins with an expression. The expression is evaluated during rule execution, and if false, the queries following are ignored. If the DRR expression is true however, the queries of the DRR are evaluated instead of the remaining queries of the rule. (Short Circuit) Each DRR expression is evaluated at the earliest point possible in the rule.

DRRs provide a mechanism to encapsulate error checking/propagation queries within rules. This eliminates the need for subordinate rules and minimizes namespace pollution. DRRs also provide a means to separate queries that are always evaluated from queries that are rarely evaluated. The expression associated with each DRR is always false unless an error occurs. So the associated queries may be interpreted rather than compiled to binary code.

The "#" parameter is distinguished in HIL and has special compilation rules. A value of zero returned through the "#" parameter indicates successful evaluation of a query. Conversely, any non-zero value returned through the "#" parameter indicates an error.

The common collection is preceded by a plus sign and contains queries that are evaluated regardless of whether the guard evaluates to true or false. The query "field.equal(0, ^#)" in the rule above for example, sets the "#" parameter to zero to indicate successful rule evaluation. This query is evaluated after the normal or abnormal collections have been evaluated. A common collection is just a shorthand notation that can be used when both the normal and abnormal collections end with the same queries.

Abstract Data Tuples

The HIL Abstract Data Tuple (ADT) is nearly identical to an OOP object. We stop short of claiming that HIL is an OOP language however, because some constructs typically found in OOP languages are not available in HIL. Furthermore, HIL is based on "code standardization" rather than "code re-use".

Presented below is a syntactically correct HIL ADT declaration called "rule_node". As shown, an ADT declaration begins with the keyword "adt". This particular ADT contains 10 data elements enclosed in square brackets and one associated rule prototype named "parse" enclosed in curly braces. Each data element is followed by a range attribute that must be the name of an ADT. The ADT names such as "string" and "parameter_node" would be user declared.

```
adt rule_node:empty
[ ADT_NAME:string,
  RULE_NAME:string,
  PARAMETERS:parameter_node,
  LOCALS:local_node,
  GUARD:query_node,
  NORMAL:query_node,
  ABNORMAL:query_node,
  COMMON:query_node,
  RESTRICTIONS:drr_node,
  NEXT:rule_node ]
{ parse(P:field, ^ROOT:rule_node, ^#:field) }.
```

Every ADT must be derived from an existing base ADT, where the derived ADT inherits all the data elements and associated rules of the base ADT. There is no "name hiding" or "method overriding" in HIL. The "rule_node" ADT above, is derived from the "empty" ADT, where the "empty" ADT is defined by default with no data elements or associated rules.

The implementation of ADTs in HIL may be encapsulated by declaring data elements and rules "private", "protected" or "public" just like OOP. ADTs may also be declared "local", "final" or "final tuple". An ADT derived from a base ADT that was declared "final tuple" may extend the base ADT with new rules but may not add new data elements. This declaration type was necessary to support array declarations.

Array ADTs

In HIL, an array is just a special type of ADT where the elements of the data tuple are referenced by position number rather than name. Presented below is the HIL declaration for a "page" array. As shown, a page is an array of 65536 "field" elements. These elements are numbered from "0" to "65535". The characters at the end of the first line declare the array "local" and "final".

```
array page:empty % $
( ELEMENT:field<65536> ).
```

A multidimensional array in HIL is just an array of arrays. For example, the HIL default "interface" ADT is an array of 16 "page" arrays as shown below.

```
array interface:empty % $
( ELEMENT:page<16> ).
```

Note that the size of an array must be defined statically and not dynamically. This is just another example of the constraints HIL imposes on the programmer to insure efficient and complete automated reasoning.

Auto-Generated Rules

Every ADT in HIL has four associated rules that are automatically generated by the compiler: "select", "modify", "equal" and "new". The first three rules are nearly identical to the three HIR basis functions with the same name. The "new" rule is used to create an instance of an ADT at runtime. Shown below is the "new" rule prototype for the "rule_node" ADT.

```
rule_node.new(^INSTANCE:rule_node, ^#:field)
```

Note that this rule will return a non-zero error value "#", if heap underflow occurs at runtime.

Indexed Rules

There is one last programming construct that needs to be presented before discussing the parallel programming capabilities of HIL; *indexed rules*. An indexed rule in HIL is just a rule with a distinguished parameter called an index. Presented just below is the prototype of an indexed rule called "main". As shown, the index parameter precedes the parameter tuple and is enclosed in square brackets.

```
rule dev.main[2](IN:line, ^OUT:line, ^#:field).
```

For a given ADT, one or more associated rules may have the same name as long as each rule of the *index set* is indexed, has an identical parameter tuple, and each index parameter number is unique. Each rule within an index set is thus interchangeable with every other rule from the same set.

An index parameter need not be instantiated until runtime; therefore a single query may be used to call any one of the rules within an index set. (late binding) This is the only form of late binding allowed in HIL because it is the only form that can be readily type checked at compile time. This is also the only form of "selection" construct in HIL. (Other than the conditional disjunction used in associated rules.)

Within any given set of index rules, the index values must begin with zero and must be contiguous. For example, a rule with index value "2" may only be defined if the index set already contains rules with index values "1" and "0". These rules may be defined in the same source file or inherited from a base ADT.

Device Oriented Programming

A *device* is just a special type of ADT evaluated under the supervision of an Operating System (OS). If the OS provides a command line interpreter, a user could conceivably type in a command such as "execute device" which directs the OS to create a thread or process from the device declaration and initiate execution. An OS could not execute a simple ADT or array ADT because the OS has no means to create a runtime process from these types of declarations.

Presented below is the declaration for a "hello" device ADT that prints the phrase "hello world" on the console screen and halts. Also presented is the definition of the associated rule "main[0]".

device hello:child

```
{ ~ main[0](IN:line, ^OUT:line, ^#:field) }
  ~ MMSG "hello, world".
```

rule hello.main[0](IN:line, ^OUT:line, ^#:field)
hello.console_out(:MMSG, ^#),
line.equal(@, ^OUT).

As shown, a device declaration begins with the keyword "device" rather than "adt". The "hello" ADT is derived from the "child" ADT that is described in the next section. "MMSG" is a *text constant* associated with the "hello" device. The rule "console_out" is inherited from the base ADT and the character "@" represents the empty pointer. Finally, the associated rule "main[0]" and the text constant "MMSG" are both declared private with the "~" character.

The associated rule "main[0]" defined above is a *constructor*. When an instance of the "hello" device is created at runtime, the OS creates a lightweight process (thread) from this declaration and initiates evaluation of the "main[0]" rule automatically. The rule "main[0]" then executes in parallel with the parent thread until it *runs to completion* or *halts*. Every device must define a constructor even if it is not needed. Additional associated rules may be defined for the index set "main" but they are not required.

The Child Default Device

Presented at the bottom of the page is a declaration for the "child" ADT that is defined by default in HIL. As shown, "child" is a device ADT because the declaration begins with the keyword "device". An instance of "child" may never actually be created however, because there is no "child.main[0]" rule defined. The "child" ADT is *abstract*. In general, any ADT derived from "child" is a device and must be declared using a device declaration.

A device may include *device parameters* in addition to the data elements, rule prototypes and constants. The "child" device has a single device parameter named "PORTS" enclosed in parenthesis. "PORTS" is an array of 16 "page" ADTs. (Note that the "interface" ADT was described previously).

The "PORTS" array is declared private with the "~" character and may only be accessed with the first four associated rules; "in_field", "out_field", "in_word" and "out_word". The "PORTS" array is two dimensional with 16 "pages" of 65536 "field" elements each. The parameters "PRT" and "POS" are used to index into the "page" and "field" arrays respectively. A "word" ADT is just an array of 16 "field" ADTs.

Device parameters are to a device what rule parameters are to a rule. They are used to pass data into and out of a device. The contents of the "PORTS" array are accessible to both the child device and the parent device. The contents are also accessible to *peer* devices that are connected to the same pages. The means to establish peer connections will be discussed shortly. For now assume that a parent device may have multiple children and these *subdevices* may communicate through shared memory.

The parent device contains an arbitrator that is automatically inserted by the compiler. The arbitrator is used to insure race free access to port pages by multiple subdevices. The associated rules "lock_ports", "unlock_ports", "wait_events" and "clear_events" are used by a child device to interact with the arbitrator of the parent device.

```
device child:empty
( ~ PORTS:interface )
{ \ in_field(PRT:field, POS:field, ^FLD:field, ^#:field),
  \ out_field(PRT:field, POS:field, FLD:field, ^#:field),
  \ in_word(PRT:field, POS:field, ^WRD:word, ^#:field),
  \ out_word(PRT:field, POS:field, WRD:word, ^#:field),
  ! halt(CODE:field, TO:line, ^FROM:line, ^#:field),
  ! lock_ports(PRT_VECTOR:field, ^#:field),
  ! unlock_ports(PRT_VECTOR:field, ^#:field),
  ! wait_events(PRT_VECTOR:field, ^#:field),
  ! clear_events(PRT_VECTOR:field, ^#:field),
  ! console_in(^FROM:line, ^#:field),
  ! console_out(TO:line, ^#:field) }.
```

Finally, the "child.halt" rule is used to halt a device and pass a *halt code* to the parent. The halt code can be a simple status value or it can be a system call to request service. The associated rule "console_out" for example, incorporates the "halt" rule to pass a string (line) to the OS for display on the console screen. The parameters "TO" and "FROM" are used to pass a string to or from the OS along with the halt code.

Port Declarations

A slightly more complicated example of a user defined device declaration; "copy", is shown below. This declaration shows the inheritance and use of, the port access/synchronization primitives. The "!" character declares the names "IN_PORT", "OUT_PORT" and "main[0]" to be "protected".

The "copy" device constructor first waits for another device or the parent to access port 0. Port 0 and 1 are then locked for exclusive access and each field of port 0 is written to the corresponding field of port 1. The port locks are then released and the device halts.

device copy:child

```
( ! IN_PORT,
  ! OUT_PORT )
{ ! main[0](IN:line, ^OUT:line, ^#:field),
  ~ transfer(POS:field, ^#:field) }.
```

```
rule copy.main[0](IN:line, ^OUT:line, ^#:field)
copy.wait_events(01000, ^#),
copy.lock_ports(03000, ^#),
copy.transfer(0, ^#),
copy.clear_events(01000, ^#),
copy.unlock_ports(^#),
line.equal(@, ^OUT).
```

```
rule copy.transfer(POS:field, ^#:field)
<FLD:field, NEXT:field>
copy.in_field(.IN_PORT, POS, ^FLD, ^#),
copy.out_field(.OUT_PORT, POS, FLD, ^#).
field.add(POS, 1, ^NEXT)
? (NEXT<0)
copy.transfer(NEXT, ^#).
```

The two names "IN_PORT" and "OUT_PORT" in the device parameter tuple of the "copy" device are *port declarations*. Starting with 0, the compiler assigns consecutive port numbers to each port name as it is encountered in the device parameter tuple. Thereafter the port name is used to access the port as shown. Ports that are not named are not accessible.

The Storage Default Device

The "child" device provides the programming primitives necessary for console and inter-device IO. This is sufficient for some small simple devices, but most user-defined devices need file IO capability to access permanent backing stores (hard disks). Shown at the bottom of this page is the device declaration for the "storage" device that is defined by default in HIL.

As shown, the "storage" device has 8 associated rules that are used to open/close files, read/write file partitions and access individual fields of file partitions. The "storage" device is derived from "child" so it inherits the device parameter "PORTS" along with all the associated rules of the "child" device.

The name "PAGES" enclosed in angle brackets is a *device local tuple* where the "directory" ADT is an array of 1024 "page" arrays. A device local tuple is to a device what the rule local tuple is to a rule. Device *local pages* of the "PAGES" array are accessible to the device but not to the parent or other peer devices.

The "open_url" rule takes a page number "PAG" as input along with a URL and mode specifier. The URL is used to access the file and the local page specified is assigned to that file. Thereafter the page number is used to access that file and that file only. The text constants "OPEN_READ", "OPEN_MODIFY" and "OPEN_NEW" are used with the "open_url" query.

Unlike other programming systems, a *file* in HIL is at most 2^{30} bits partitioned into 1024 *partitions* of 2^{20} bits each. The "read_page" and "write_page" rules are used to copy a file *partition* to or from the backing store to the local page. The four "get_xxx" and "set_xxx" rules are then used to access individual "field" elements of the local page.

```
device storage:child
< ~ PAGES:directory >
{ ! open_url(PAG:field, URL:line, MODE:field, ^#:field),
  ! read_page(PAG:field, PARTITION:field, ^#:field),
  ! write_page(PAG:field, PARTITION:field, ^#:field),
  \ get_field(PAG:field, POS:field, ^FLD:field, ^#:field),
  \ set_field(PAG:field, POS:field, FLD:field, ^#:field),
  \ get_word(PAG:field, POS:field, ^WRD:word, ^#:field),
  \ set_word(PAG:field, POS:field, WRD:word, ^#:field),
  ! close_url(PAG:field, ^#:field) }
! OPEN_READ = 0,
! OPEN_MODIFY = 1,
! OPEN_NEW = 3.
```

The "cp" device declaration presented below is an example of a device derived from the "storage" device. The "cp" device opens a file named "old_file", reads in the first partition to local page 0, opens another file "new_file" and writes the contents of page 0 to the new file. "OLD" and "NEW" are text constants (strings) associated with the "cp" device.

```

device cp:storage
  < ~ PAGE >
  { ~ main[0](IN:line, ^OUT:line, ^#:field) }
  ~ OLD      "old_file",
  ~ NEW      "new_file".

rule cp.main[0](IN:line, ^OUT:line, ^#:field)
  cp.open_url(.PAGE, :OLD, .OPEN_READ, ^#),
  cp.read_page(.PAGE, 0, ^#),
  cp.close_url(.PAGE, ^#),
  cp.open_url(.PAGE, :NEW, .OPEN_NEW, ^#),
  cp.write_page(.PAGE, 0, ^#),
  cp.close_url(.PAGE, ^#),
  line.equal(@, ^OUT).

```

The name "PAGE" in the device local tuple above is a *page declaration*. Like port declarations, starting with 0, the compiler assigns consecutive page numbers to page names as they are encountered in the device local tuple. Unlike port names however, page names are not required to access device local pages. There are 1024 pages in the local array "PAGES" and it would be cumbersome to require that all local pages be named prior to use.

The Parent Default Device

The "child" and "storage" default devices provide all the inter-device, console IO and file IO primitives needed to build conventional non-parallel programs. The HSSM architecture is specifically targeted at massively parallel software problems however, and *composition primitives* are needed to compose new devices from collections of existing devices. The "parent" device provides these primitives.

Presented at the bottom of this page is the device declaration for the "parent" device that is defined by default in HIL. As shown, the "parent" device is derived from "storage" so it inherits all the inter-device, console IO and file IO primitives of "storage" and "child".

The "parent" device also extends the device local tuple of "storage" with a single element called "CHILDREN". This element has a range attribute of "subdevices" which is an array of 16 "child" ADTs. The "CHILDREN" array is declared private with the "~" character and its contents may only be accessed with the first 10 associated rules.

The programming model for the "CHILDREN" array is analogous to a bank of 16 read/write CD-RW drives that are interconnected. You can *load* any of the 16 drives with a CD-RW disk. Once loaded, the disk drive will automatically access the first track of the CD. Thereafter, CD access can be *paused* and *resumed*, or halted entirely and the CD *unloaded*. Once paused the disk drive can be *restarted* on another track.

The "load_device" rule takes as input, a URL and subdevice number "DEV". The file associated with the URL is accessed and the device declaration contained there is used to create a runtime process. A pointer to the process is then inserted into the "CHILDREN" array using the subdevice number. Thereafter, the subdevice number is used to control the subdevice.

The "INDEX" parameter of the "restart_device" rule is used to select a rule from the "main" index set of the subdevice during restart. The "IN" and "OUT" parameters correspond to the parameters with the same name in the parameter tuple of a "main" rule. Likewise the "TO", "FROM" and "CODE" parameters match the parameters with the same name in the "halt" rule.

Finally, the "wait_halt" rule is used to context switch from user mode to supervisor mode and the "lock_page" and "unlock_page" rules are used to guarantee race-free access to device local pages when these pages are shared with child subdevices.

```

device parent:storage
  < ~ CHILDREN:subdevices >
  { ! load_device(DEV:field, URL:line, IN:line, ^#:field),
    ! restart_device(DEV:field, INDEX:field, IN:line, ^#:field),
    ! resume_device(DEV:field, ^#:field),
    ! pause_device(DEV:field, ^#:field),
    ! unload_device(DEV:field, ^#:field),
    ! get_out(DEV:field, ^OUT:line, ^#:field),
    ! get_error(DEV:field, ^ERROR:field, ^#:field),
    ! get_to(DEV:field, ^TO:line, ^#:field),
    ! set_from(DEV:field, FROM:line, ^#:field),
    ! get_code(DEV:field, ^CODE:field, ^#:field),
    ! wait_halt(D_BITS:field, ^S_BITS:field, ^#:field),
    ! lock_page(PAG:field, ^#:field),
    ! unlock_page(PAG:field, ^#:field) }.

```

The device declaration for a "copy2" device is presented below as an example of an ADT derived from the "parent" device. The "copy2" device begins by creating two subdevices using the "copy" device declaration previously discussed. It is assumed that this device declaration is contained in the file "copy.hil".

The "copy2" device then locks its local page 0 and reads in a file partition from a file whose name was input from the parent of "copy2". Unlocking of local page 0 then triggers the first subdevice to begin execution and make a copy of local page 0 in local page 1. This then triggers the execution of the second subdevice that makes yet another copy in local page 2. The parent "copy2" device waits on each of its subdevices to halt before unloading them.

device copy2:parent

```
< ~ DEV0:copy(0, 1),
  ~ DEV1:copy(1, 2) >
{ ~ main[0](IN:line, ^OUT:line, ^#:field) }
  FILENAME "copy.hil".
```

rule copy2.main[0](IN:line, ^OUT:line, ^#:field)

```
<X:field, Y:field>
copy2.load_device(.DEV0, :FILENAME, ^#),
copy2.load_device(.DEV1, :FILENAME, ^#),
copy2.lock_page(0, ^#),
copy2.open_url(0, IN, .OPEN_READ, ^#),
copy2.read_page(0, 0, ^#),
copy2.close_url(0, ^#),
copy2.unlock_page(0, ^#),
copy2.wait_halt(01000, ^X, ^#),
copy2.get_error(.DEV0, ^#, ^#),
copy2.wait_halt(02000, ^Y, ^#),
copy2.get_error(.DEV1, ^#, ^#),
copy2.unload_device(.DEV0, ^#),
copy2.unload_device(.DEV1, ^#).
```

The device local tuple of "copy2" declared above contains two *subdevice declarations*; "DEV0" and "DEV1". As shown, a subdevice declaration consists of a *subdevice name*, a range attribute and a *net list*. Like port declarations and page declarations, the compiler assigns subdevice numbers to subdevice names as subdevice declarations are encountered. The range of a subdevice name must be the name of a device.

A net list is a mapping from the parent local pages to the ports of each subdevice. This becomes a little more obvious when a *device schematic diagram* is used to represent a parent device. Presented in Figure 8 is the device schematic diagram of the "copy2" device declared above. As shown, a device is represented as a notched square with 16 pins. Each pin represents a device port and pins are numbered from 0 to 15 beginning at the notched corner and proceeding clockwise. Device local pages are drawn as circles and

connections between local pages, subdevice ports and device ports are represented by lines (nets).

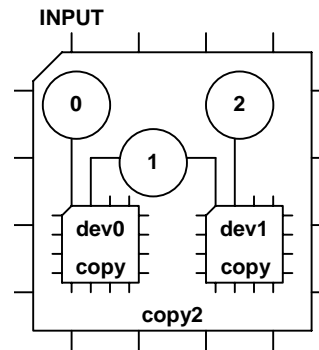


Figure 8 Schematic Diagram of "copy2" Device

Comparing Figure 8 with Figure 3, it should be noted that a parent device represents the conclusion of a conjunctive collection. Each subdevice represents a query in the hypothesis of the same conjunctive collection and local pages are just complex shared variables. There could also be a subdevice to represent the "main" index set of the parent device.

Page declarations precede subdevice declarations when both exist in the device local tuple and page names may be used in net lists to represent pages. Port names may also be used to represent device port pages in net lists. Thus the ports of a parent device may be mapped to the ports of subdevices. Presented below is a device declaration for a "copy3" device that uses page names and port names in a net list. Figure 9 shows the schematic diagram for this device.

device copy3:parent

```
( ~ INPUT )
< ~ PAG0,
  ~ PAG1,
  ~ DEV0:copy(.INPUT, .PAG0),
  ~ DEV1:copy(.PAG0, .PAG1) >
{ ~ main[0](IN:line, ^OUT:line, ^#:field) }
  FILENAME "copy.hil".
```

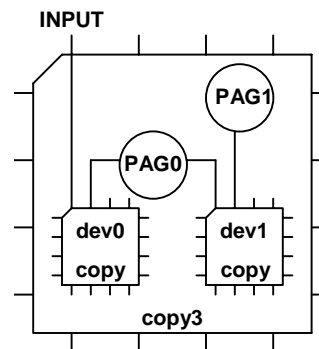


Figure 9 Schematic Diagram of "copy3" Device

Summary

Presented just below in Figure 10 is a Venn diagram of an HSSM compiler. As shown, the HSSM IR is the intersection of the parser and code generator. The HSSM IR also intersects the "Automated Reasoning Unit". In the future, we will be designing massively parallel software, and designs of this scope will require CAD tools that are far more sophisticated than the tools currently available. These tools will necessarily be built upon formal reasoning systems and this implies that an IR must have a simple formal mathematical interpretation to be of practical use.

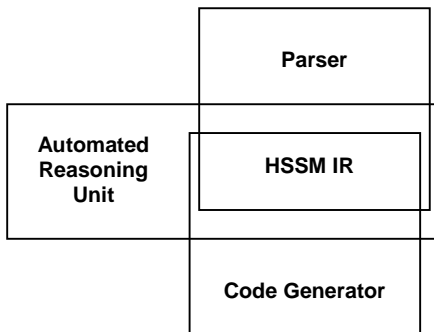


Figure 10 Venn Diagram of a Compiler

The HSSM IR is a relational algebra, derived from the first order predicate calculus. It is powerful enough to represent the most significant programming constructs of an OOP language yet is still simple enough to present to first year CS students. The HSSM IR is also robust enough to cleanly represent large parallel programs as a hierarchical composition of successively simpler entities.

Further Reading

Anyone with an undergraduate degree in computer science or mathematics should have the background to understand the derivation of the HIR presented in this paper. The undergraduate textbook "Logic and its Applications" by Burke and Foxley[1] is an excellent reference if needed.

The textbook "Artificial Intelligence - A Modern Approach" by Russel and Norvig[2] also covers most of the same material but expands the discussion to include automated reasoning and the entire field of artificial intelligence. This material will give the reader some idea of the application domain of HIL and the HSSM programming model. A device is actually a crude form of *intelligent agent*.

HIL and the HSSM IR were not developed in a vacuum. Both are actually components of a larger macro-architecture design project that includes not only language and mathematical technology, but also hardware micro-architecture and a distributed operating system.

The documentation for this design project; "The HSSM Manual"[3] contains the derivation of the HSSM IR, a specification for the HIL and the specification for a file interchange format; the HIF. A link to the current version of this manual, (2.0) in pdf format, may be found at:

http://hssm_manual.home.att.net

A link to the version 0.0 of this manual[4] may also be found at the same sight. This first version of the manual contains the derivation of the HSSM IR and a specification for the HSSM Virtual Machine (HVM). Where the HVM is a hardware micro-architecture designed specifically for the HSSM IR.

The HVM supports native execution of both conditional disjunction statements and device oriented parallel programming. This first version however, did not include support for Dynamic Range Restrictions. It was also assumed that message passing would be used as a communication mechanism between peer devices. The HVM is now being redesigned to match the current version of HIL. Version 3.0 of the manual will include this redesign. An overview of the HVM, version 0.0, was published in the April 2002 edition of "SIGPLAN Notices"[5].

Beyond the HVM redesign, a micro-kernel for a distributed operating system will follow. The device oriented programming model of HSSM requires that each parent device incorporate a supervisor level micro-kernel. This micro-kernel is inserted automatically by the compiler; thus the compiler code generator cannot be completed until the OS design is complete.

References:

- [1] E. Burke and E. Foxley, *Logic and its Applications*, Prentice Hall Europe, 1996. ISBN 0-13-030263-5
- [2] S. Russel and P. Norvig, *Artificial Intelligence A Modern Approach*, Prentice Hall, 1995. ISBN 0-13-103805-2
- [3] P.D. Wells, "The HSSM Manual", Ver 2.0, http://home.att.net/~hssm_manual/Master.pdf, January 30, 2004.
- [4] P.D. Wells, "The HSSM Manual", Ver 0.0, http://home.att.net/~hssm_manual/hssm_manual.pdf, October 29, 2001.
- [5] P.D. Wells, "The HSSM Macro-Architecture, Virtual Machine and H languages", ACM SIGPLAN Notices, 37(4), April 2002:74-82.