Orangepath HPR H2 User Manual
Very First Tentative Draft

August 20, 2009

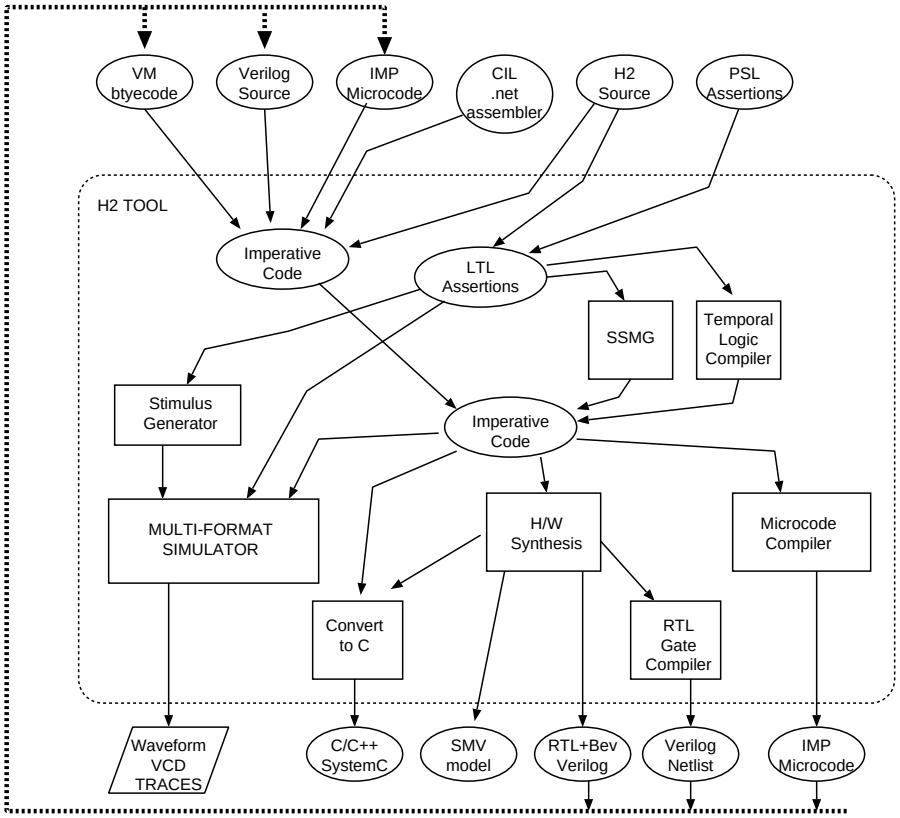

Figure 1: The flows implemented in the current tool.

0.1 Overview Summary

Orangepath is a refinement framework designed for synthesis of protocols and interfaces in hardware and software forms.

Orangepath HPR represents designs an an hierarchy of abstract machines. Each machine is a database of declarations, executable code and goals. The goals are assertions about the system behaviour, input directly, or generated from compilation of temporal logic and data conservation rules into automata. Executable code can pass through the system unchanged, but any undriven internal nodes are provided with driver code that ensures the system meets its goals.

The H2 language possesses various subsets and the intention is that these may be freely mixed, at a fine level, to describe a design. The main subsets are structural, temporal logic (PSL), C-like imperative and sysML-like state hierarchic charts.

0.2 Tool Flow

The current H2 tool reads in various inputs forms, some of which can be outputs from previous compilations. Figure 1 shows the available flows. This figure shows

that inputs are aggregated into a pair of rule sets: executable rules and assertion rules. Naming and scoping rules for the identifiers are preserved from the input structures. Not shown in the figure are several bypass flows, where the outputs can be fed into the internal simulator instead of having to be fed into a simulation in a subsequent run of the tool.

Executable rules are held as an executable VM bytecode for an HPR virtual machine. This is readily compiled to IMP microcode form for the reference H2 microprocessor, SystemC or to Verilog in various forms, including netlists and custom VLIW processors. SMV output is also an option, for feeding to the nuSMV model checker.

Executable rules may also be fed to the internal simulator, called diosim, where they can be executed with stimulus generation form a PRBS (psudo-random binary sequence) generator or from stimulus read from 'plant' files.

Assertions can be compiled to executable form in various ways or else checked during diosim simulation. Rather than being checked, they can constrain pseudo-random input sequences used in simulation.

The Temporal Logic Compiler operates on PSL (property specification language) assertions and SERES (sugar extended regular expressions) to generate executable automata for synthesis or simulation.

The SSMG component is described in a separate document and is a complete subproject with respect to H2 HPR. SSMG is the main refinement component that converts assertions to executable logic using goal-directed search.

CIL .net input language is processed by a free-standing front end called kiwic. This has its own manual that shares some text with this H2 manual. CIL code is the assembly language used by the mono and .net projects. The HPR tool can read in CIL assembly code when invoked using the `kiwic` command.

The H2 input format includes finite state machine definition in SysML statechart format.

The Orangepath H2 compiler accepts inputs in a variety of forms and generates an executable design that is implemented as a mixture of hardware and software. The input may be non-specific in terms of the resources to be targeted and may be non-specific in terms of the number of states and state transitions of the output machine, whereas the output always amounts to a deterministic automaton. Hence the compiler is making decisions over hardware/software partitioning and details of the algorithm.

The input is any mixture of RTL, assertions, declarations and imperative software and the output is a mixture of hardware and software components. The following, named steps are used:

1. **Input Step(s):** Read in each input form and store to disc or hold internally as an H2 Machine.

2. **Flatten Step:** Combine all H2 machines into a single hierarchic H2 machine, known as the source machine, generating instance names for each child H2 machine so that all variable names have a unique path from the design root.

3. **Refinement Step:** Convert the source machine into the target machine by rewriting and augmenting the executable rules such that they are deterministic and consistent with the assertion rules.

4. **Partition Step:** Convert the target machine into a number of communicating sections, where each consists of RTL (register transfer level) or IMP (imperative) code. Save each section as a separate output file.

5. **Compile step:** Convert each IMP section into either hardware or software, either generating an RTL or a micro-controller (MPU) section and write to disk as a `.s` microcode file or `.vnl` Verilog net list. C and SystemC output formats can also be used for the.

6. **Simulate step:** Optionally, simulate the collection of RTL and MPU sections together with stimulus and plant files provided by the user. The assertion rules are monitored during the simulation and coverage is logged.

The most important step in the compiler is the refinement step that converts the source H2 machine to the target H2 machine. The H2 machine is the most important data structure and we describe this first, before describing the input and output forms.

0.3 H2 Machine

All input forms are converted into a common internal structure which consists of a tree of H2 machines. An H2 machine is recursively defined in that any H2 machine can contain a number of child H2 machines.

An H2 machines is a pen-tuple that consists of a pair of disjoint lists of variables, a list of sub-machines, a list of assertion rules and a list of executable rule blocks. All lists are unordered. The connections to sub-machines, the assertions and the basic executable rules over the variables in union of the two variable lists. The first list contains variables visible to a parent H2 machine whereas the contents of the second are local to the current machine.

Variables have three basic types: parameter, value or event. A value or parameter variable ranges over a finite range of integers and where this is just the range 0..1 we call it a boolean variable. An event variable ranges of any finite enumeration but also possesses the property of not currently occurring, which may be thought of as an extra value. Another type is the mutex, which is a special form of boolean value type with special properties.

Not all of the variables may be needed in the generated target machine, but parameter variables explicitly must not occur in the target. All parameter variables must be eliminated during compilation through not being needed or by being given a constant value, either specified by the user or chosen by the compiler.

Expressions occurring in the assertion and executable rule blocks range over the variables, future values of variables and a special non-deterministic symbol, called non-det, denoted with a query in parenthesis `'(?) '`.

Future values of variables are denoted with the circle or X operator. The future value of a parameter is itself. The future value of an external input to the system must not be used, since this is not causal.

Value variables that are only updated when some event occurs are called *sequential* variables. Sequential variables that are updated only by the occurrence of common event are part of a *clock domain* related to that event. All other value variables are called *combinational* variables.

New values for value and event variables are defined by the executable rules contained in the rule blocks. The aggregate of executable rules must never try to assign more than one value to a variable at once, ie. it be consistent.

Executable rules are assignments held in rule blocks. A rule block consists of an optional guard and an SP. A rule block with no guard is called a combinational rule block. A sequential rule block is guarded by an event expression. An SP is recursively defined as either an assignment of a variable from an expression, or a sequential list of SPs, or a parallel list of SPs. The order of listing members of a sequential SP is important whereas it has no significance for a parallel SP. The order is important for a sequential SP since updates to variables from executing one member of the list are experienced by the next in the list as it evaluates its assignment expression(s). For the parallel SP, there is no visibility of changes made by one member at another member: changes are nominally held over until the end of the parallel SP is reached, and then they are merged. It must be statically determinable that there are no inconsistencies in the merge.

A rule block is called *compilable* if it can be converted to a normal form where there are no powers of X present, except for a power of unity on the left hand side of each sequential executable rule.

Various other normal forms for the executable rules exist, and procedures to convert between them exist, but some procedures have exponential cost and are avoided unless needed. A normal form where every variable is updated in just one rule block can readily be converted to a hardware model in RTL Verilog and/or VHDL. A normal form where there are no parallel SPs in a block allows ready conversion of that block to a basic block in a block-structured imperative output format, such as C.

One way to represent input language forms that use a thread that blocks at various places is to convert each of the resume points to a separate executable rule block. This technique is used for the H2 bevblock construct.

Every assertion rule is either a safety, liveness or initial assertion.

The next value operator: circle 'o '. $o\ e \stackrel{\Delta}{=} X(e) \stackrel{\Delta}{=} X(e,1)$.

0.4 Input Formats

The main input format is source files in the H2 language and .net CIL but Verilog RTL input format and IMP machine code are also supported. A separate user manual describes the Kiwi CIL input format.

The H2 input format includes finite state machine definition in SysML statechart format and regular expressions.

The H2 language is in flux, so check the h2grammar.yy yacc file and see the examples for details.

0.4.1 CIL input format

The CIL .net assembly code is generated by a large number of third party compilers from various input languages. Please see the separate kiwic manual for details of this input format.

Chapter 1

H2 Syntax

The H2 language possesses various subsets and the intention is that these may be freely mixed, at a fine level, to describe a design. The main subsets are structural, temporal logic (PSL), C-like imperative and sysML-like state hierarchic charts.

1.0.2 Concrete syntax tree

The H2 HPR concrete syntax tree is following yacc file:

1.0.3 Abstract syntax tree

The H2 HPR abstract syntax tree is defined using the following SML datatype:

1.1 H2 Types

All expressions either have integer type or else denote partial types that are eliminated during an elaboration phase of compilation, such as channel names, parameters and module instance names.

The boolean type is represented by the integer subrange 0..1;

Type expressions are a type name, an enumeration, an integer range or a vector range.

An integer range is two integers separated by two dots, such as '`0..1`'. The first number is, by convention, lower than the second, but the order is ignored.

A Verilog RTL style vector range is two integers in brackets separated by a colon, such as '`[7:0]`'. This example defines an integer range of 0..255 and names the bits of a bus representation of the integer.

An array type is denoted with square brackets and if the length of the array is known, this is placed in the brackets. The subscript range of the array is from zero to the number in brackets minus one.

An enumeration type is a set of constant strings. It is defined in braces, prefixed with the keyword `enum`. For example

```
enum { Play, Forward, Stop, Reverse }
```

All enumeration constants must be disjoint within their scope of use.

Note: stategraphs implicitly define enumerations with their state names.

Type names can be established with the `typedef` statement. For example

```
typedef safe_range_t = -55..55;
typedef transport_t = enum { Play, Forward, Stop, Reverse };
```

1.2 H2 Expressions

1.2.1 H2 Constant Expressions

The following constants are builtin: `X`, `(?)`, `true`, `false`. The symbol 'X' denotes don't care.

Any sequence of digits is a base-ten integer, and such integers can start with a minus sign.

1.2.2 H2 Variables, Events and Parameters

Variables are defined using the `node` statement, or one of its shorthand forms (§1.6.2).

Variables have three basic types: parameter, value or event. A value or parameter variable ranges over a finite range of integers and where this is just the range 0..1 we call it a boolean variable. An event variable ranges of any finite enumeration but also possesses the property of not currently occurring, which may be thought of as an extra value. Another type is the mutex, which is a special form of boolean value type with special properties.

All variables have a scope that is the facet they are defined in and all facets directly instantiated below, unless textually masked by more closely enclosing declarations.

Access between facets is enabled using path names consisting of a facet instance names separated by dots.

An facet definition may contain a list of structural formal parameters in parenthesis after its name. These are formal parameters to the facet and are expanded at instantiation using call by name. They are used only for structural (macro-style) elaboration and are not user variables.

All parameter variables must be eliminated during compilation through not being needed or by being given a constant value, either specified by the user or chosen by the compiler.

1.2.3 Operator Expressions

The symbol 'X', when standing along, denotes don't care. When used as a function it denotes the next state operator. The expressions $X(e)$ is short for $X(e,1)$ and means the next value of expressions e. Higher values of n in $X(e,n)$ denote further values into the future, using the expansion $X(e,n+1) = X(X(e,n))$.

Bit extract is denoted with brackets: eg. '`e[e]`'.

The diadic pling operator, '`c!e`', writes a value to a named channel. The expansion of channel writes is explained in §1.4.

1.2.4 Function Application

Function applications are either of built-in functions or of user functions that act as macros and are expanded at compile time.

The built-in function '`pause()`' is used to denote a bus-settling delay or memory-barrier. By calling this function, the current thread is bocked until all writes made to variables or nets are flushed out and made visible to other processes. It should always be given the argument 1.

The built-in function '`hpr_testandset(mutex, bool)`' must take a variable of type mutex (a boolean subtype of value) as its first argument. If the second argument is 'true' then the function attempts to set the mutex and returns the previous value. If the second argument is 'false' then the mutex is cleared and false is returned.

The built-in function '`print()`' causes console output under simulation or on an embedded platform if supported. The arguments are converted to an ASCII representation and output in turn.

The built-in function '`exit(rc, [msg ...])`' causes a simulation to exit. An error is indicated using a non-zero return code in the first argument. Supporting message information may also be provided in subsequent arguments. The behaviour on embedded platforms, if supported, is to halt execution of all threads until a reset occurs.

The built-in function 'X' is explained in§1.2.3.

User-defined functions are declared with the `fundef` keyword.

1.3 H2 Assertions

An assertion statement constrains the behaviour of the system.

The assertion statements may be free standing, or may be used in the C-like code or in the action section of a statecharts.

Where free standing, they must universally hold.

Only the safety assertions and fairness marker can occur in the C-like code and statecharts. Their meaning is then respectively guarded by the C-like thread reaching them or the state being active.

Assertions refering to events and patterns of events follow the syntax and semantics follows PSL: [['Property Specification Language Reference Manual' Version 1.1 June 9, 2004]

```
assertion ::=
    always   [ <string> : ] <pslexp>;
|   never    [ <string> : ] <pslexp>;
|   initial  [ <string> : ] <pslexp>;
|   live     [ <string> : ] <pslexp>;
|   fair     [ <string> : ] <pslexp>;
```

1.4 H2 Channels

The compiler implements message passing channels. The pling operator is used to put a value to a channel and the query operator is used for reading from a channel.

Both are blocking operators, because channels implement reliable flow-control.

The channel operations may be used in the C-like code or in the action section of a statecharts. A blocking channel operation in a statechart will make the whole of/part of the state machine block (TODO explain).

The current implementation of channels is via straightforward macro expansion in the front end of the compiler. Channels are implemented by shared access to entries in an array called C

Both operators make copies of the channel designator on entry, in case the user's expression should change while blocked, and the write operator makes a copy of the value to be sent. Copies are not made for manifest constant expressions. Copies are kept in fresh variables denoted below with the '_c' suffix. The allocation of index values to the array is handled by the compiler, and the back-end compilation phase replaces hardware constant indexes with scalars.

The expansion of the write operation *c*!*e* is

```
c_c = c;
e_c = e;
waituntil !C.ack[c_c];
C.data[c_c] = e_c;
C.req[c_c] = 1;
waituntil C.ack[c_c];
C.req[c_c] = 0;
```

The expansion of the read operation ?*c* is

```
c_c = c;
waituntil C.req[c_c];
r_c = C.data[c_c];
C.ack[c_c] = 1;
waituntil !C.req[c_c];
C.ack[c_c] = 0;
return r_c;
```

1.5 H2 C-like Imperative Statements

The H2 language possesses various subsets. The main subsets are structural, temporal logic (PSL), C-like imperative and sysML-like state hierarchic charts. This section defines the C-like imperative subset.

H2 includes a typical block-structured imperative programming language with semantics based on those of C, but extended with operators including channel write and the '`guard`' statement.

1.5.1 The H2 `if` statement

```
if (<exp>) <statment>
if (<exp>) <statment> else <statment>
```

The H2 `if` statement executes its argument if the condition evaluates to a non-zero value.

1.5.2 The H2 `while` statement

```
while (<exp>) <statment>
```

The H2 `while` statement evaluates its body while its argument evaluates to a non-zero value.

1.5.3 The H2 `emit` statement

```
emit  <var>;
```

The H2 `emit` statement sends a nullary event to a named variable that must be an event variable. Parameterised events are not generated using this statment. Instead they are generated by assigning values to the event variable.

1.5.4 The H2 `waituntil` statement

```
wait (<exp>);
```

The H2 `waituntil` statement takes an expression and blocks a thread until the expression would evaluate to a non-zero value.

1.5.5 The H2 `wait` statement

```
wait (<exp>);
```

The H2 `wait` statement takes a postive numeric argument and blocks a thread for that number of time units.

1.5.6 The H2 `guard` statement

```
guard (<exp>) <statment>
```

The H2 `guard` statement evaluates its body if the guard expression gives a non-zero value, but the thread exits immediately from the body if the guard expression becomes zero at any time during the execution of the body, whether the thread is blocked or not.

1.5.7 The H2 `resultis` statement

```
resultis (<exp>);
```

The H2 `resultis` statement returns a value to a surrounding context. It is intended to be used with the `valof` operator. It is interchangeable with the `return` statement.

1.5.8 The H2 `return` statement

```
return <exp>;
```

The H2 `return` statement returns a value to a surrounding context. It is intended to be used in function bodies. It is interchangeable with the `resultis` statement.

1.5.9 The H2 `skip` statement

```
skip;
```

The H2 `skip` statement does nothing.

1.5.10 The H2 `continue` statement

```
continue;
```

The H2 `continue` statement transfers execution to the head of the innermost surrounding `while` or `for` loop.

1.5.11 The H2 `break` statement

```
break;
```

The H2 `break` statement transfers execution to the exit point of the innermost surrounding `while` or `for` loop.

1.5.12 The H2 label statement

```
L:
```

The label statement defines a target for a `goto` statement.

1.5.13 The H2 `goto` statement

.

```
goto L;
```

The H2 `goto` statement transfers execution to the named label which must be present somewhere in the same behavioural sequence.

1.5.14 The H2 block statement

.

```
{ S1 S2 .... Snnn }
```

The H2 block statement consists of any number of statements enclosed inside braces and they are executed in sequence.

1.5.15 The H2 assignment statement

. .

```
<variable> = <exp>;
```

The H2 assignment statement assigns a value to a variable. The assignment is actually an expression and any expression can be used in this context. A function call expression becomes a procedure call in this way.

1.5.16 The H2 procedure call statement

.

```
<name>(<arg1>, ...);
```

The H2 procedure call statement executes a procedure, including certain builtin procedures (§1.2.4). The call is actually an expression and any expression can be used in this context. A function call expression becomes a procedure call in this way.

1.5.17 The H2 channel write statement

.

```
<channel> ! <exp>;
```

The H2 channel write statement evaluates an expression and writes the value to a named channel. The expression is evaluated immediately but the thread can then become blocked if the is not ready to read.

1.6 H2 Structural Statements

The H2 language possesses various subsets. The main subsets are structural, temporal logic (PSL), C-like imperative and sysML-like state hierarchic charts. This section defines the structural subset.

The structural statements in H2 are unlike those in most languages. The power of H2 is in its structural statements.

1.6.1 Facet definitions

An H2 program consists of a number of facet definitions. A facet definition includes instances of other facets and local code. Facet definitions may be builtin, loaded from libraries or user defined. An instance of a facet is commonly called a node and the `node` statement, or a shorthand for it, is used to instantiate all facets, except for the topmost facet. The topmost facet is instead mentioned on the command line and is called the root.

The facet declaration syntax is:

```
structural_item ::=
     <directional_context> |
     <node_declaration> |
     <constant_defintion> |
     <behavioural_section> |
     <statechart> |
     <connect_statement> |
     <assertion>

facet_definition ::=
   <facet_type> <facet_name>  [ (<structural_formal>, ... ) ]
   {
      [ <structural_item> ... ]
```

```
    }

structural_formal ::= <id>
```

The facet_type must (currently) be one of the builtin facet types: section, unit, protocol or interface.

The facet_name must be a fresh identifier.

The structural formal parameters are optional, but must all be provided with values whenever the facet is instantiated. They bind identifiers occurring in the facet and there is almost no restriction over the connect of the identifiers (e.g. types, facet names, constants etc.). Currently, the topmost facet cannot have formals since it is instantiated from the command line.

The contents of the facet are structural items defined in the rest of this section.

H2 defines a number of builtin leaf facets: `node`, `protocol`, `action`, `section`, `interface` and `section`. The node is the most general facet and also the most basic: a node can be as simple as a boolean variable, but it can be any other variable, channel or facet. The user defines his own facets that inherit from one of these. Facets are heirarchic, in that each may instantiate further, lower facets. Each instantiation may be forwards or reversed and inverted or not.

1.6.2 Node declaration statement

The H2 node statement declares

```
node_declaration ::=
    node    <node_type> [ <modifier> ... ] : <nid> [ , <nid> ... ];

nid ::=  <id> | !<id> | <id> (<exp> [, <exp> ... ])
node_type ::= <id>
```

The `node` statement creates one or more named instances of an facet. The names must be disjoint. Every node has a type. The type name can be used instead of the keyword ‘node’ where the type is one of these builtin facet types: channel, event, mutex, parameter, section, protocol, action, unit or interface.

When nodes are instantiated, modifiers may be used. The available modifiers are: `out`, `in`, `inout`, `event`, `unsigned`, , `signed`, `channel`, `parameter`, `initiator`, `target`, `forward` , `reverse` and range declarations for arrays and scalars of fixed ranges.

Nodes may be declared as inverted and/or reversed. An inverted declaration is made by placing the pling character before the identifier. A reversed declaration is made by inserting the `reverse` modifier in the modifier list. Nodes may be

parameterised by supplying one or more arguments after the identifier. The number of expressions must match the number of formal parameters in the node type definition.

The program is elaborated in the textual order it occurs in the file. All identifiers ultimately form a single, flat name space. At any point in the file, all identifiers already defined through being a facet directly or indirectly instantiated in the current facet are in scope. Multiple identifiers of the same name may be in scope at once: e.g. when there are two instances of a given sort of interface. Where multiple identifiers of the same name are in scope at once, it is an error to refer to one of the multiply-defined identifiers in an ambiguous way. Sufficient facet prefix path details must be supplied.

Range and Array modifiers

A modifier of the form `[ n .. m ]` defines a node that can take on an integer range of values.

A modifier of the form `[ h : l ]` defines a packed vector node with high and low bit positions called h and l. This is another way of defining a range. A value of zero is normally used for l.

A modifier of the form `[ n ]` defines an array with n locations, indexed from zero.

A modifier of the form `[ ]` defines an array with an unbounded number of locations.

Only one of the first two forms is allowed for a given node.

Forwards, Reverse and Neutral modifiers

```
directional_context ::= forward: | neutral: | reverse:
```

When connecting a pair of components, the inputs of one component are normally connected to outputs of the other, and vice versa. This requires that these terms must be reversed when a specification written for one side of the interface is being used at the other. This is known as having a *handed* pair. To overcome this issue, when any type of node is declared/instantiated, including interface nodes and complete sections, it is defined in a directional context. A reverse context causes inputs to be interpreted as output and outputs to be interpreted as inputs.

The default directional context is forward, but neutral and reverse contexts also exist. The current directional context is altered to the desired value by the `forward:`, `reverse:` and `neutral:` labels. These labels alter the current context in the textually following declarations until another label is encountered. In addition, the first two of these three words may also appear as a modifier in the actual declaration of the node. The overall context of a node is reverse if there is an odd number

of reversings in the referring path. A path is reversed by each reverse instantiation. A reverse instantiation either contains the reverse keyword as a modifier or is inside a reverse directional context, but not both. If it both, they cancel out. Declarations inside a neutral directional context are not altered and have their default meaning regardless of how many reversings there are on the referring path.

1.7 Temporal Regular Expressions

The H2 language possesses various subsets. The main subsets are structural, temporal logic (PSL), C-like imperative and sysML-like state hierarchic charts. This section defines the temporal logic (PSL) definitions.

TODO. Contents of this section are missing, but are mainly bog standard PSL.

1.8 Stategraph Definition

The H2 language possesses various subsets. The main subsets are structural, temporal logic (PSL), C-like imperative and sysML-like state hierarchic charts. This section defines the statechart subset.

The stategraph (or statechart) defines a finite state machine, where each state has a state name. A top-level stategraph is always active, meaning it is in exactly one state. On the other hand, a stategraph that is instanced as a child stategraph within a state in another stategraph is inactive (not in any state) unless its parent is in that instantiating state. A state may instantiate any number of child stategraphs but recursion is not allowed.

The stategraph general form is:

```
stategraph graph_name()
{

  state statename0 (subgraph_name, subgraph_entry_state), ... :

        entry:  statement;
        exit:   statement;
        body: statement;

        statement;
        ...             // implied 'body:' statements
        statement;

        c1 -> statename1: statement;
        c2 -> statename2: statement;
```

```
        c3 -> exit(good);
        ...

        exit(good) -> statename3: statement;
        exit(bad) -> statename4: statement;

        ...


   endstate


   state statename2:
...
...
   endstate


   state abort: // A special state that can be
                // forced remotely (also called disable).

   ...

}
```

A state may contain tagged statements, each of which may be a basic block if required. They are distinguished using three tag words. The ‘entry’ statement is run on entry to the state and the ‘exit’ statement is run on exit. The ‘body’ statement is run while in the state. A ‘body’ statement must contain idempotent code, so that there is no concept of the number of times it is run while in the state. Statements with no tag are treated as body tagged statements. Multiple occurrences of statements with the same tag are allowed and these are evaluated as though executed in the textual order they occur or else in parallel (current implementation is serial but this will be change to parallel, so watch out!).

A state contains transition definitions that define the successor states. Each transition consists of a boolean guard expression, the name of one of the states in the current stategraph and an optional statement to be executed when taking the transition. In situations where multiple guard expressions currently hold, the first holding transition is taken.

The guard expressions range over the inputs to the stategraph, which are the variables and events in the current textual scope, and the exit labels of child stategraphs.

When a child stategraph becomes active, it will start in the starting state name is given as an argument to the instantiation, or the first state of no starting name is

given.

A child stategraph becomes inactive when its parent transitions, even if the transition is to the current state, in which case the child stategraph becomes inactive and active again and so transitions to the appropriate entry state.

A child stategraph can cause its parent to transition when the child transitions to an exit state. There may be any number, including zero, of exit states in a child stategraph but never any in a top-level stategraph. The parent must define one or more transitions to be taken for all possible exit transitions of its children. An exit state is either called 'exit' or 'exit(id)' where 'id' is an exit tag identifier. Exit tags used in the children must all be matched by transitions in the parent, or else the parent must transition itself under the remaining exit conditions of the child or else the parent must provide an untagged exit that is used by default.

A stategraph may be wholly enclosed inside any conditional statement, such as an '`if`' or '`case`' statement, in which case it is as though all of its internal activity is guarded by that condition: the condition is simply folded inside every construct to the point where a conditional is allowed. The stategraph does not reset to its starting state when this guard does not hold.

A stategraph with a state called `abort` may be disabled from elsewhere in the same bundle using the '`abort`' statement. Please see §1.8.1.

The stategraph general form is sufficient to encompass the SysML state machines.

1.8.1 Abort Statement

The '`abort`' statement is used for a remote abort of a stategraph.

Syntax:

```
if (g) abort stategraph_name1, stategraph_name2, ...;
```

The abort statement must be conditional, otherwise the stategraph would never leave its abort state, and the abort guard, *g* may either be an event or level expression. When the abort guard is a level expression it takes precedence over any transitions in the stategraph that lead from the abort state.

```
if (g) abort stategraph_name1, stategraph_name2, ...;
```

Chapter 2

Joining Automata Synthesis

The contents of this chapter describe a particular research project and should be ignored by general users (at the moment).

2.0.2 Meaningful Play and Mitre Automata

The valid operation of a protocol is defined in terms of the operations it performs on its interface. When a protocol performs an operation, we say it makes a *play*. Only a small number of plays may be valid at any one time, as dictated by an automata that transitions on each operation or in other constraints (eg. a wire that is already low cannot go low). A *play* is an operation performed on the interface by the protocol in a given state of the constraining automata.

The *meaningful play set*, or just play set, is a subset of the plays that convey information. Other plays are artifacts of the protocol that can be modified or ignored without changing the the meaning of the information conveyed.

A pair of interfaces must be connected to each other for information to flow. The wiring, logic or code used to connect the interfaces is called the connection. A valid connection between a pair of interfaces can be defined by an automata, the *mitre automata*. This automata has a pair of inputs that range over the play sets from each interface. Every state of the mitre automata is an accepting state, but the connection must be designed so that mitre automata never gets stuck. In general, a single mitre automata can interconnect more than just two interfaces. Also, more than one mitre automata can be specified, where all operate in parallel and every play is always accepted at all automata at once.

The H2 `play` statement is a prefix to any other behavioural statement. It denotes that the operation(s) performed by the statement, in the current state of the executing thread or stategraph, is/are a member of the meaningful play set. An identifier, the *play name*, may be assigned to the play statement, postfixed by a colon. A formal parameter list may also be specified.

A play statement is an annotation and has no semantic effect on its argument,

which is always executed as normal when it is run.

```
play_statement ::= play [ <play_id> [ <formals> ] : ]  <behavioural_statement>
```

Here is a typical example, where a play called 'mysend' is defined. The behavioural statement is a block containing three successive imperative statements. The formal parameter list is simply a further annotation that denotes which variables occurring in the behavioural statement convey run-time data. These are handled symbolically during mitring whereas the remainder are given concrete values.

```
play mysend(dout) : { if_dout = dout; pause(); strobe = 1; }
```

Meaningful plays always occur in pairs, where one half of the pair is executed by each side of the interface. This is called a *rendezvous*.

The mitre automaton can be defined using any H2 form of expression, prefixed with the keyword `mitre`. For instance, it can be defined as a statechart (§1.8) or using a behavioural section (§2.0.3). Play names can be used inside a mitre definition as though they were imperative statements. They can also be prefixed with the `left` and `right` keywords or a facet instance name (identifier). Side-effecting statements, such as assignments to variables must not be used inside a mitre definition, except to local variables used only in the mitre definition, such as for encoding state.

```
play_occurrence_statements ::=
    <play_id>;
  | left           : <play_id>;
  | right          : <play_id>;
  | <facet_id>     : <play_id>;
```

The `left` and `right` qualifiers are optional play name prefixes that cause reference to either the first or second argument to the connect statement, respectively. When a prefix is left out and the same play name occurs on both sides, such as when connecting a pair of instances of the same interface, then the play applies to both sides at once.

Where more than one mitre automata is defined, their product is implied: that is, they are all logically running at once and none must ever get stuck.

Mitre examples using behavioural sections

Where both sides of an interface only have one meaningful play and it is called `foo` it is sufficient to write

```
mitre while (1) { foo; }
```

Where we wish to make a ping on one side do a pong on the other, it is sufficient to write

```
mitre while (1) { left ping; right pong; }
mitre while (1) { right ping; left pong; }
```

2.0.3 Behavioural section

A behavioural section contains any number of H2 behavioural statements (§1.5). They are executed as though enclosed in a `while(1) { ... }` infinite loop.

```
behavioural_section ::=
    {
      <behavioural_statement> ...
    }
```

2.0.4 Assertion

An assertion statement constrains the behaviour of the system.

Assertions refering to events and patterns of events follow the syntax and semantics follows PSL: [['Property Specification Language Reference Manual' Version 1.1 June 9, 2004]

```
assertion ::=
    always   [ <string> : ] <pslexp>;
|   never    [ <string> : ] <pslexp>;
|   initial  [ <string> : ] <pslexp>;
|   live     [ <string> : ] <pslexp>;
|   fair     [ <string> : ] <pslexp>;
```

2.0.5 Connect Declarations

Connections are declared with the connect statement. The H2 connect statement joins two or more facets, either directly or by generating glue logic and/or glue code. The facets are denoted with heirarchic path expressions (separated with dots). A connection has an optional name and if more that two facets are to be joined by one connection, each must have a local facet instance identifier.

```
connect_statement ::= connect [ <connection_id> : ] <exp>, <exp>
         [ mitre [<flaglist> : ] <structural_item> ] ;
connect_statement ::= connect [ <connection_id> : ] <facet_id> : <exp>, ...
         [ mitre [<flaglist> : ] <structural_item> ] ;
flaglist ::=  [ <flag_id>=<flag_exp>, ... ]
```

When the connection identifier is supplied, it is used as a name for the connection and as the root name for any instantiated or generated code. A connection identifier can be specified as the rendering root (§**??**) for compilation, which allows the synthesised code to be captured to output files (VNL, microcode, C++, and so on).

The `mitre` keyword introduces a mitre automata to be used in the connection implementation (§2.0.2).

When only two facets are to be joined, they need not be given facet instance identifiers because the built-in names `left` and `right` are used by default. Facet instance identifiers used in a connect statement are local nicknames, private to that connection and may only be used inside the mitre clause. Where the facet in question is also instantiated as part of the generated design, it will have a primary instance name from that instantiation.

Where a mitre automata is not present, a simple connect is implemented, where outputs from one facet are matched with similarly-named inputs of other sides and wired together.

Where a mitre automata is defined, it is multiplied with the interface automata listed in the connect statements. The state space is then collapsed over the various rendezvous designated with play annotations so that no part of a same-named play on different automata happens in separation from the others of that name. Finally, a maximal live manifold is selected that contains the idle states from all automata and all also all of the rendezvous. A live manifold is an automata that consists of states and edges from the collapsed product machine where all states are reachable from all others. A maximal live manifold is a manifold where as many paths as possible are included (however, there can be local minima problems). An H2 machine is then generated that connects up the participating facets in a way that implements the manifold. Where desired live paths are not included in the manifold, the user can constrain the selection either by adding assertions into the facet definitions or using a facet as the mitre automata and putting assertions in that.

Where the structural item after the mitre keyword defines more than one finite state machine, their product machine is first formed and then the connection is built as before. The resulting interface obeys all mitres at once.

The behaviour of the generated interface logic can be modified by specifying flag expressions. A number of flag identifiers exist that can be set to constant values in the flag list. However, whether the interface logic consists of hardware gates, software code or some mix is not altered by these flags: that is instead selected by the normal H2 synthesis option flags (§**??**).

The `reset` flag may be used to specify a reset input or condition to the interface logic. The reset flag expression may range over nets occurring in any facet of the interface or otherwise undefined variables which is/are thereby defined as auxiliary inputs to the interface.

The `clock` flag may be used to specify a clock signal for the interface logic. It may refer to any binary signal occurring in any facet of the interface or to an otherwise undefined variable which is thereby defined as an auxiliary input to the interface. The clock flag is ignored if the output mode is to generate entirely software. When no clock flag is specified, asynchronous logic is generated.

Chapter 3

SSMG Refinement Algorithm

H2 is a vehicle for exploring various refinement algorithms.

In [**?**], it is proposed that all interfaces are constructed using a combination of elemental interface paradigms and that any description or implementation of an interface can be processed to be represented in this way. The processing is a form of parsing that generated a so-called interface transform. Once an interface is represented this way, it can be render in a variety of detailed output styles.

The default refinement algorithm uses a depth-first search and has exponential cost in the worst case. A SAT-based algorithm was also explored in the paper [**?**], but is disabled in the tool by default.

The refinement algorithm must first find a subset of the rules and variables that are possibly needed in the target machine. The following steps achieve this.

1. Identify, from the compiler command line, the top-level target variables that are to be driven by the target machine, thereby creating a first target variable set, thenceforth known as the current target variable set. Create an empty set of rules called the current rule set.

2. Identify any executable rules that drive variables in the target variable set, or their past or future values, or assertion rules that refer to them or their past or future values. Add these safety and executable rules to the current rule set.

3. If any variables occur in the current rule set that are not external inputs and are not members of the current target variable set, add them to the target variable set and go back to previous step.

The refinement algorithm then proceeds to generate further executable rules from the assertion rules and to fill in concrete values for the parameters and values of (?) encountered, thereby generating a deterministic target machine.

The refinement algorithm uses a CNF/clause representation of the design and is based around a built-in SAT solver.

1. The safety assertions are all first converted into a conjunctive-normal form and held on a safety clause list.

2. For any executable rule that assigns a value to the next state of any variable, v, all occurrences of $X(v,n)$ where $n >= 2$ are substituted for using that executable rule.

3. For all values of all external inputs, subject to plant constraints, the executable rules are examined for consistency and any parameter values or non-det transitions that would make the executable rules inconsistent are noted. Where only one possible value for a parameter exists, the parameter is substituted out with that value, otherwise the constraints are added as additional clauses to the safety clause list.

4. If the safety clause list is non-empty, a clause with a minimal number of un-driven variables in its support is removed and converted to an executable rule where one of the variables is driven by a LUT function of all inputs and driven variables where the LUT contains fresh parameters. Go back to previous step.

5. When the safety clause list is empty, select a setting of all parameters in the executable rules that creates a finite state machine that satisfies all the liveness assertion rules. If none can be found, then backtrack to the previous step and select a different free variable of a safety clause to be driven by a LUT.

6. Partition the resulting machine into hardware and software components and output.

This algorithm does not reflect the sequencing constraints of the plant...

cf. Take all at once and SAT solve!

Chapter 4

Transactor Synthesis

The command line flag `-xtor` invokes the transactor synthesis refinement algorithm. There are some example on the web site.

TODO: describe it more.

Chapter 5

Orangepath Synthesis Engines

The Orangepath project supports various internal synthesis engines. The aim is to include SSMG but some more simple engines are also provided. The other engines include the FSM generator, the PSL compiler and the restructurer.

Because all input is converted to the HPR machine and all output is from that internal form it is sensible to use the HPR library for translation purposes without doing any actual synthesis.

A synthesis engine rewrites one HPR machine as another.

5.1 A* Live Path Interface Synthesiser

The H2 front end tool allows access to the live path interface synthesiser.

The A* version is described on this web page. http://www.cl.cam.ac.uk/ djg11/wwwhpr/gpibpage.html

The follow-on to this work is being undertaken by MJ Nam.

5.2 Transactor Synthesiser

The transactor synthesiser is described on this link

http://www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/transactors

5.3 Asynchronous Logic Synthesiser

The H1 tool implements an asynchronous logic synthesiser described on this link.

http://www.cl.cam.ac.uk/ djg11/wwwhpr/dsasynch.html

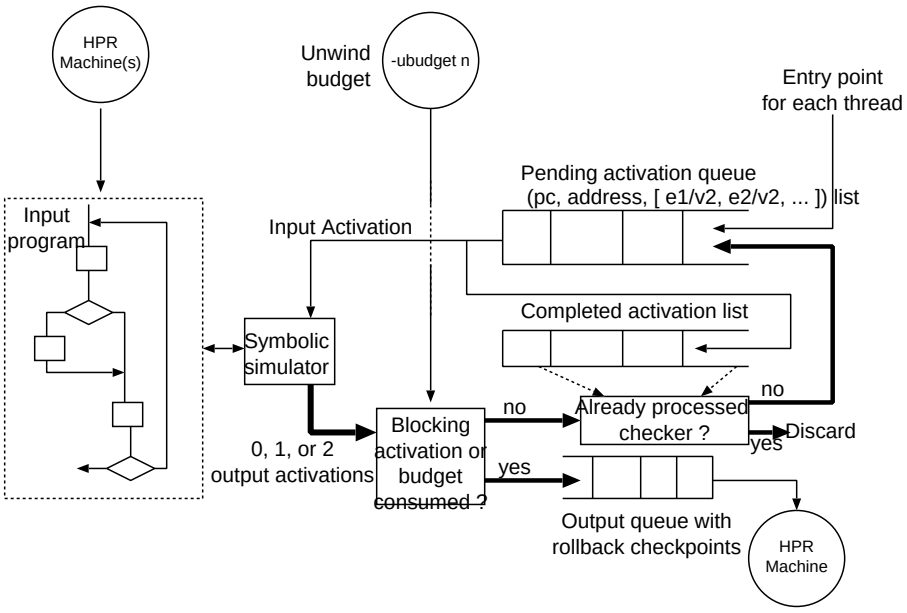

Figure 5.1: The Synchronous FSM generator in the Orangepath tool.

5.4 SAT-based Logic Synthesiser

The H1 tool implements a SAT-based logic synthesiser described on this link.

http://www.cl.cam.ac.uk/ djg11/wwwhpr/dslogic.html

(This synthesiser is currently not part of the main HPR revision control branch.)

5.5 Synchronous FSM Synthesiser

The HPR tool contains a synthesiser/generator for synchronous FSMs that converts a program in the HPR imperative language in to a finite state machine. The language contains assignments, conditional gotos, fork/join and leaf calls to HPR library functions.

The input and output to the FSM generation process are HPR machines. The output machine uses the so-called XRTL style that is readily converted to Verilog RTL by a subsequent stage.

An additional input, from the command line, is an unwind budget: a number of basic blocks to consider in any loop unwind operation. Where loops are nested or fork in flow of control, the budget is divided amongs the various ways.

5.5.1 Synthcontrol

Minor changes in the operation of FSM synthesiser are controlled with the `-synthcontrol` command line option. The sequencer for a thread can be unpacked, normal or one-hot. Unpacked is selected with `sequencer:unpacked`.

Minor changes in the operation of FSM synthesiser are controlled with the `-synthcontrol` command line option. The sequencer for a thread can be unpacked, normal or one-hot.

The string `preserve-sequencer` should be supplied to keep the per-thread vestigal sequencer in RTL output structures. This makes the output code more readable but can make it less compact for synthesis, depending on the capabilites of the FPGA tools to do their own minimisation.

The string `resets:synchronous` should be passed in to introduce synchronous resets to the generated sequencer logic. This is the default.

The string `resets:asynchronous` should be passed in to introduce assynchronous resets to the generated sequencer logic.

The string `resets:none` should be passed in to supress reset logic for FPGA targets. FPGA's tend to have built-in, dedicated reset wiring.

```
-synthcontrol 'preserve-sequencer;resets:none;sequencer:packed'
```

The central data structure is the pending activation queue, where an activation consists of a program counter name, program counter value and environment mapping variables that have so far been changed to their new (symbolic) values.

The output is a list of finite-state-machine edges that are finally placed inside a single HPR parallel construct. The edges have to forms (g, v, e) (g, fname, [args]) where the first form assigns e to v when g holds and the second calls function fname when g holds.

Both the pending activation queue and the output list have checkpoint annotations so that edges generated during a failed attempt at a loop unwind can be discarded.

The pending activation list is initialised with the entry points for each thread. Operation removes one activation and symbolically steps it through a basic block of the program code, at which time zero, one or two activations are returned. These are either added to the output list or to the pending activation list. An exit statement terminates the activation and a basic block terminating in a conditional branch returns two activations. A basic block is terminated with a single activation at a blocking native call, such as hpr_pause. When returned from the symbolic simulator, the activation may be flagged as blocking, in which case it is fed to the output queue. Otherwise, if the unwind budget is not used up the resulting activations are added to the pending queue.

A third queue records successfully processed activations. Activations are discarded and not added to the pending queue if they have already been successfully

processed. Checking this requires comparison of symbolic environments. These are kept in a ”close to normal form” form so that syntactic equivalence can be used. This list is also subject to rollback.

Operation continues until the pending activation queue is empty. A powerful proof engine for comparing activations would enable this condition to be checked more fully and avoid untermination with a greater number of designs.

5.6 PSL Synthesiser

The PSL synthesiser converts PSL temporal assertions into FSM-based runtime monitors.

5.7 Statechart Synthesiser

The Sys-ML statechart synthesiser is built in to the front end of the H2 tool. It must be built in to other front ends that generate HPR VMs,

5.8 SSMG Synthesiser

SSMG is the main refinement component that converts assertions to executable logic using goal-directed search. The SSMG synthesiser is described in a separate document and is a complete sub-project with respect to HPR.

5.9 Restructure Synthesiser

The RTL-style machines can be restructured, so that different operations occur in different cycles, with automatic insertion of holding registers to maintain data values that would not be available when needed.

Restructuring is need to avoid structural hazards arising when an ALU or multiplier is not fully-pipeline or when a memory has insufficient ports for the level of concurrent access required.

Chapter 6

Output Formats

The HPR library contains a number of output code generators. All of these write out a representation of an internal HPR machine. Not all forms of HPR machine can be written out in all output forms, but, where this is not possible, a synthesis engine should be available that can be applied to the internal HPR machine to convert it.

Certaint output formats can encode both an RTL/hardware-style and a software/threaded style. For instance, a C-like input file can be rendered out again in threaded C style, or as a list of non-blocking assignments using the SystemC library.

The following output formats may be created:

1. **RTL Form:** The RTL output is written as a Verilog RTL. One module is created that either contains just the RTL portion of the design, or the RTL and instances of each MPU that is executing software parts of the design.
2. **Netlist Form:** The RTL output is compiled to a structural netlist in Verilog that contains nothing but gate and flip-flop instances.
3. **H2 IMP Form:** The HPR form is output to an IMP file. This has the same syntax as the imperative subset of H2.
4. **SMV form:** The HPR VM is output as an SMV code and the assertions that have not been compiled or refined are output as assertions for SMV to check.
5. **C Form:** The HPR VM is output as C code suitable for third-party compilers. RTL forms may also be output as synthesisable SystemC.
6. **UIA MPU Form:** The IMP imperative language is compiled to IMP assembly language and output as a `.s` file.
7. **IP XACT form:** The structural components are written out as IP XACT definitions and instances.

8. **S-expression form:** The HPR VM is dumped a lisp S-expression to a file.

9. **UIA Machine Code:** The IMP assembly is compiled to machine code for the UIA microcontroller. This is output as Intel Hex and also as a list of Verilog assignments for initialising a memory with this code.

The net-based output architecture is suitable for direct implementation as a custom SoC (system on chip). H2 defines its own microcontroller and we use the term MPU to denote an H2 microcontroller with an associated firmware ROM. The net-based architecture consists of RTL logic and some number of MPUs. However, by requesting that all output is as C code for a single MPU, the net-based output degenerates to a single file of portable C code.

Additional output files include log files and synthesisable and high-level models of the UISA microprocessor that executes IMP machine machine code.

Index