



STREP – IST 033709

VERTIGO

---

Verification & Validation of Embedded System Design  
Workbench

## Deliverable D2.2 Tool for integrating modelling flow

DUE DATE 31 May 2007  
ACTUAL DATE 11 Jun 2007  
START OF PROJECT 01 June 2006 DURATION 30 months  
ABSTRACT This document provides a summary of activities related to the implementation of strategies and tools for the integration of the VERTIGO modelling flow.  
AUTHOR, COMPANY G.Pravadelli, N.Bombieri, G.DiGuglielmo, F.Fummi, F.Stefanni – University of Verona  
J.Raik, A.Chepurov – University of Tallinn  
D.Karlsson, Z.Peng – University of Linköping  
WORKPACKAGE/TASK WP2  
FILING CODE VERTIGO/Deliverables/07\_D2.2\_VERTIGO-R2  
KEYWORDS modelling, hardware description languages, property specification languages

### DOCUMENT HISTORY

Release	Date	Reason of change	Status	Distribution
1	21/05/2007	First draft	Draft	Internal
2	28/05/2007	Integration of contribution	Draft	Internal

D2.2: TOOL FOR INTEGRATING MODELLING FLOW  
VERTIGO/DELIVERABLES/07\_D2.2\_VERTIGO-R3

IST 033709 VERTIGO

31 MAY 2007

3 11/06/2007

Final

Internal + EC

# Table of Contents

---

<b>1.</b>	<b>Introduction.....</b>	<b>1</b>
<b>2.</b>	<b>HIFSuite overview.....</b>	<b>2</b>
2.1	The HDL intermediate format (HIF).....	3
2.2	VERTIGO's technology integration flow .....	3
<b>3.</b>	<b>SC2HIF .....</b>	<b>5</b>
3.1	SC2HIF architecture.....	5
3.1.1	Sc2AST .....	6
3.1.2	RM_Comments.....	6
3.1.3	ScXML2HIF .....	6
3.2	Extensions to AIF .....	6
3.3	Supported constructs .....	7
3.4	Compiling SC2HIF .....	10
3.5	Running SC2HIF .....	10
3.6	Bug list .....	11
<b>4.</b>	<b>VHDL2HIF .....</b>	<b>14</b>
4.1	Supported constructs .....	14
4.2	Compiling VHDL2HIF .....	15
4.3	Running VHDL2HIF .....	15
4.4	Bug list .....	15
<b>5.</b>	<b>HIF2HDL.....</b>	<b>19</b>
5.1	Supported constructs .....	19
5.2	Compiling HIF2HDL.....	20
5.3	Running HIF2HDL.....	20

5.4	Bug list .....	20
<b>6.</b>	<b>EFSM generation and manipulation via HIF .....</b>	<b>21</b>
6.1	Compiling Phase1 .....	21
6.2	Running Phase1 .....	22
6.3	Bug list .....	22
<b>7.</b>	<b>HLDD generation and manipulation via HIF.....</b>	<b>24</b>
7.1	HLDD generation .....	24
7.1.1	Running Phase1 to generate HLDD.....	24
7.2	HLDD translation into AGM format.....	25
7.2.1	Installing the HLDD2AGM tool .....	25
7.2.2	Running HLDD2AGM translation.....	25
7.2.3	Bug list .....	25
<b>8.</b>	<b>PRES+ generation and manipulation via HIF.....</b>	<b>26</b>
8.1	Translation overview .....	26
8.2	Supported constructs .....	27
8.3	Compiling HIF2PRES .....	27
8.4	Running HIF2PRES .....	28
<b>9.</b>	<b>Property management via HIF .....</b>	<b>29</b>
9.1	Setting up and running FoCs.....	30
9.2	VHDL checkers .....	30
9.3	Checker conversion via HIFSuite .....	31
9.4	SystemC checkers .....	32
<b>10.</b>	<b>A mini hackers guide for HIF tools.....</b>	<b>34</b>
10.1	General tips .....	34
10.2	The RxApp library .....	34

10.3	The HIF library.....	34
10.3.1	HOWTO extend the HIF Library - Guidelines .....	35
10.4	HIF2HDL.....	35
10.4.1	HOWTO extend HIF2HDL - Guidelines.....	36
<b>11.</b>	<b>References.....</b>	<b>37</b>

# 1. Introduction

---

The rapid development of modern embedded systems requires the use of flexible tools that allow designers and verification engineers to efficiently manipulate HDL descriptions throughout the design and verification steps by using different tools and methodologies.

Nowadays, it is common practice to define new systems by reusing previously developed components, that can be possibly modelled at different abstraction levels (TLM, RTL, etc.) by means of different hardware description languages (HDLs) like VHDL, SystemC, Verilog, etc.. Moreover, the new models are verified by using static and dynamic tools that may adopt different input languages and different paradigms (e.g., finite state machines, decision diagrams, Petri nets, ...).

Such an heterogeneity requires either using co-simulation and co-verification techniques [1], or convert different HDL pieces of code into a homogeneous description [2]. However, co-simulation techniques slow down the overall simulation, while manual conversion from an HDL representation to another, as well as manual abstraction/refinement from an abstraction level to another, are not valuable solutions, since they are error-prone and time consuming activities. Thus, both co-simulation and manual refinement reduce the advantages provided by the adoption of a reuse-based design and verification methodology.

This deliverable, in the context of the VERTIGO project, presents *HIFSuite*, i.e., a closely integrated set of tools and APIs developed during task T2.6 of work package WP2 to allow system designers to manipulate HW/SW descriptions in a uniform and efficient way.

## 2. HIFSuite overview

This section briefly introduces *HIFSuite*, the intermediate format *HIF* upon which *HIFSuite* relies, and the related integration flow that has been defined to allow project partners to easily integrate the several tools and methodologies developed (or under development) in the VERTIGO project.

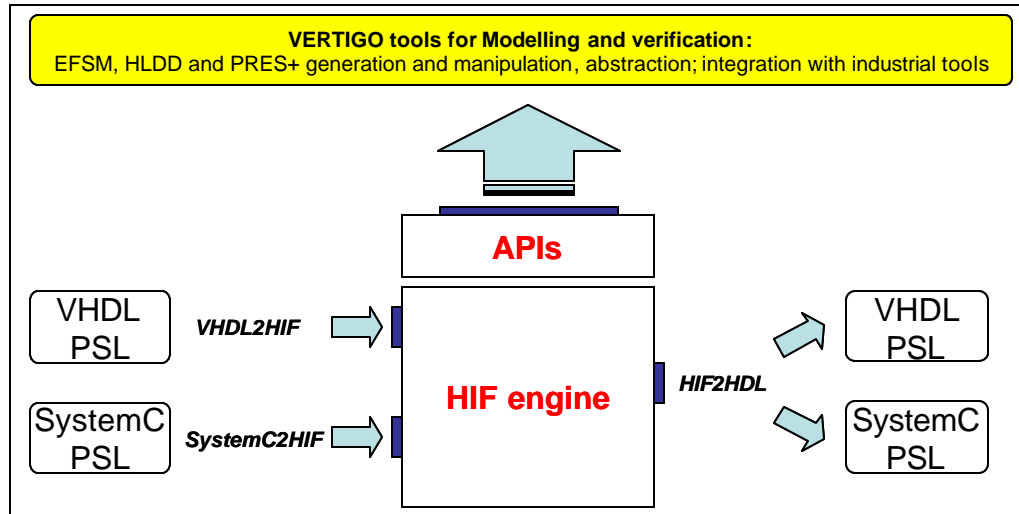


Figure 1 HIF-based tool integration.

The main features of *HIFSuite* are depicted in Figure 1. *HIFSuite* allows designers and verification engineers to:

- analyze and parse VHDL or SystemC descriptions representing HW/SW systems or derived from Property Specification Language (PSL) properties;
- extract HIF representations of the parsed descriptions;
- manipulate and instrument the HIF representations by using HIF-based applications;
- define their own HIF-based manipulation tools by exploiting a powerful set of application programming interfaces (APIs);
- generate new VHDL or SystemC descriptions that reflect the changes introduced by the manipulation of the HIF representation;
- define their own conversion tools for supporting other hardware description languages.

The core of the suite is the HIF engine which is used for representing HDL constructs. The front-end tools *SC2HIF* and *VHDL2HIF* allow converting, respectively, SystemC and VHDL descriptions into HIF models, while the back-end tool *HIF2HDL* allows converting HIF models into VHDL or SystemC descriptions.

Once the HIF representation has been obtained, a powerful set of APIs can be used for visiting and manipulating the HIF code. Currently, several tools have been developing in VERTIGO by exploiting such APIs. Some tools are intended to aid verification like, for example, the fault injector, others are related to abstraction and refinement steps, like the RTL to TLM abstractor and the transactor generator, others are useful to make the modelling flow inside VERTIGO uniform, like the EFSM/HLDD/PRES+ generators.

The next sections focus on the HIF format and the set of tools developed for unifying the VERTIGO modelling and verification flow.



Industrial tools, like for example *imPROVE-HDL* and *Assertains*, do not directly manipulate HIF code, but the integration among them and academic tools is obtained via VHDL and PSL. In particular, academic tools can read SystemC or VHDL code, convert it into the HIF format, manipulate the HIF description and then generate VHDL code suited for industrial tools.

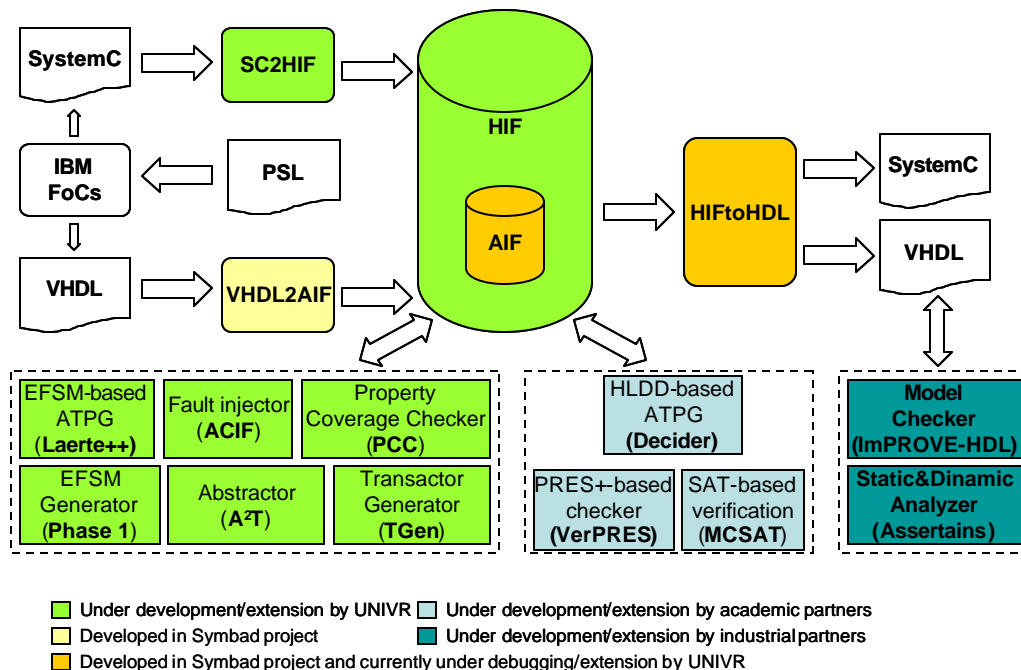


Figure 3: The role of HIF in unifying the language infrastructure.

### 3. SC2HIF

This section presents the tool *SC2HIF* developed by UNIVR, which translates SystemC code into HIF code. The section describes also how to compile and run *SC2HIF*, which constructs are supported, and how AIF has been extended into HIF to support SystemC constructs.

#### 3.1 SC2HIF architecture

The general architecture of *SC2HIF* is described in Figure 4.

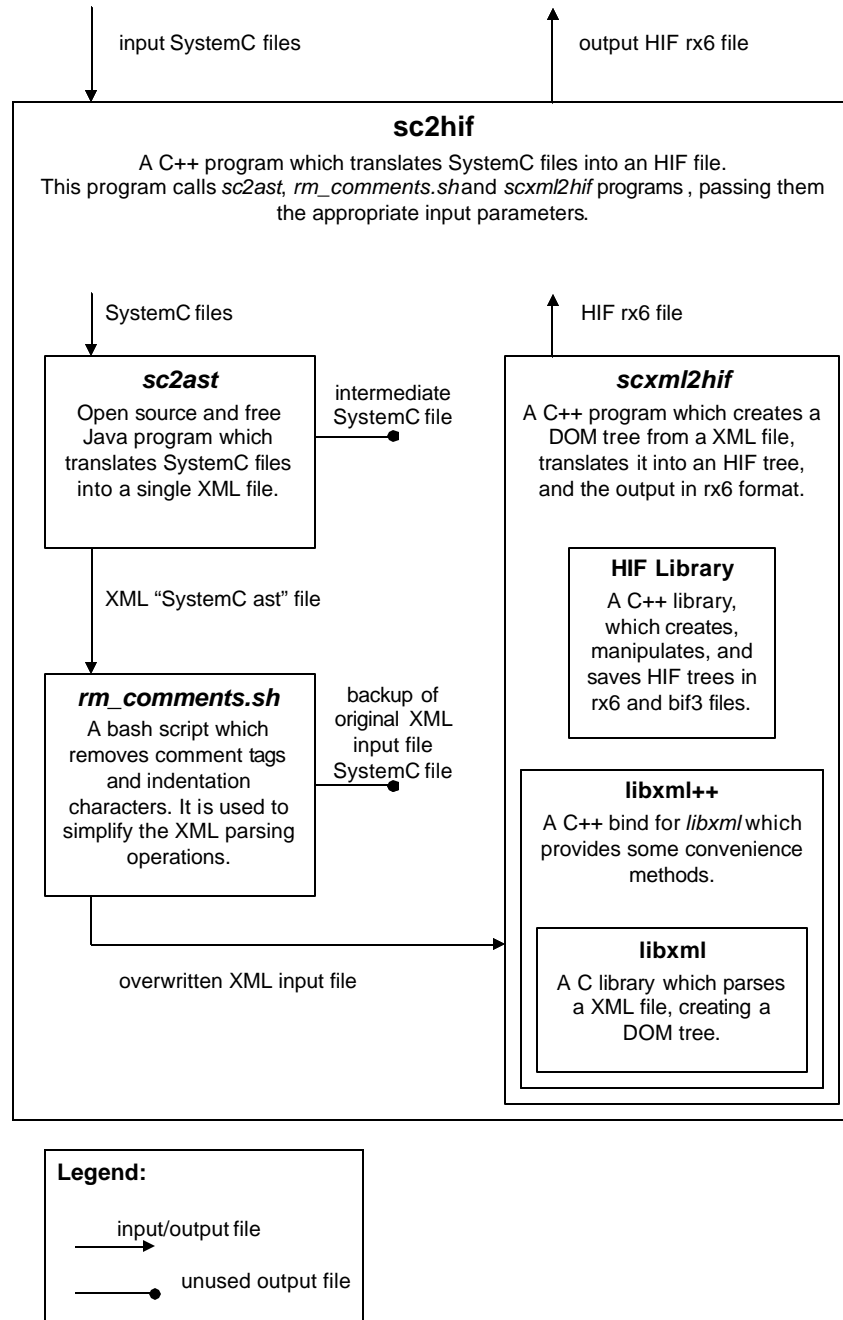


Figure 4: SC2HIF architecture.

*SC2HIF* is a C++ program which calls in sequence the following programs:

1. *Sc2Ast*
2. *RM\_Comments*
3. *ScXML2Hif*

Each program is described in detail in the following Subsections .

### 3.1.1 Sc2AST

*Sc2AST* is a open source free Java program which translates SystemC input files into a unique XML file, that describes the abstract syntax tree (AST) corresponding to the SystemC code.

Two output files are created:

1. an intermediate SystemC file, which contains a copy of all input files with some pre-processing feature;
2. the XML file describing the AST.

The XML file is provided as input to *RM\_Comments*.

### 3.1.2 RM\_Comments

*RM\_Comments* is a bash script which removes all "COMMENT" tags, all blank lines, and useless spaces and tabs from the input XML file. The original input file is backup, and the output is overwritten into the input file.

The overwritten XML file is provided as input to *ScXML2HIF*.

### 3.1.3 ScXML2HIF

*ScXML2HIF* generates an HIF output file from an XML input file. The input file must describe the System C AST, without any "COMMENT" tag, space or tab between XML tags.

*SCXML2HIF* uses three libraries:

- *HIF Library*: a C++ library that describes all HIF objects, and provides methods to read or save HIF trees.
- *LibXML++*: an open source free C++ library that is a wrapper and a binder for C++ programs of the *LibXML* library. *SCXML2AST* uses this library to create a DOM tree.
- *LibXML*: an open source free C library, which can read XML files with both DOM and SAX parsers.

A file, called *sc2ast\_grammar.html*, describes the grammar which is used by *Sc2Ast*. It has been used as a guideline about how to parse the input XML.

## 3.2 Extensions to AIF

---

The target of the AIF library developed during the Symbad project is VHDL, thus AIF is not able to support all the SystemC constructs. This Section describes extensions and changes implemented by UNIVR to extend AIF into HIF for supporting SystemC.

- **SC\_METHOD, SC\_THREAD, SC\_CTHREAD**. AIF supports only SC\_METHODs that are translated into a state table (*STOject*). In order to support also threads, a new property has been created (*PROPERTY\_STOJECT\_KIND*). The default value for the property is "SC\_METHOD". Other possible values are: "SC\_THREAD" and "SC\_CTHREAD".
- **WAIT**. The AIF library includes a *WaitObject* object which is insufficient to support all kinds of *wait* instructions implemented in SystemC. In fact, *WaitObject* accepts only a

single *ValueObject* as parameter. On the contrary, the new HIF *WaitObject* includes a list of *ValueObjects*. Moreover, it supports a property (*PROPERTY\_WAIT\_KIND*) to specify the kind of the *wait*, i.e., the *wait()* method ("WAIT") or the *wait\_until()* method ("WAIT\_UNTIL").

- **METHOD.** AIF supports only functions and procedures, but SystemC have also instance methods. HIF supports instance methods like special cases of procedures and functions, by using the first argument as instance object. A new property (*PROPERTY\_METHOD\_KIND*) has been specified to allow HIF tools to recognize the difference between instance methods and simple procedures/functions. The default value of such a property is "NOT\_INSTANCE", while the other possible value is "INSTANCE".
- **REFERENCES.** AIF does not support references. HIF supports references, via a specific property (*PROPERTY\_DECLARATION\_IS\_REFERENCE*), which can be set as a declaration object property. Two values are allowed for such a property: "FALSE", which is the default value, and "TRUE".
- **STREAMS.** AIF supports only normal functions, and has no way to understand that a "shift operator" syntax can hide a stream. HIF supports streams as normal methods, which are associated to the property *PROPERTY\_METHOD\_IS\_STREAM*. Two values are allowed for such a property: "FALSE", which is the default value, and "TRUE".
- **SPECIFIERS.** AIF does not support type specifiers like: *static*, *auto*, *extern*, *mutable*, *register* and *volatile*. A new property (*PROPERTY\_SPECIFIER*) has been added to allow HIF-based tools to recognize specifiers. Possible values for such a property are: "PROPERTY\_STATIC", "PROPERTY\_AUTO", "PROPERTY\_REGISTER", "PROPERTY\_EXTERN", "PROPERTY\_MUTABLE", "PROPERTY\_VOLATILE".
- **BIGINTEGER.** AIF accepts big integers but it uses the C *int* data type to store its value, thus loosing all bits that cannot be handled by an integer. A new class (*BigintvalObject*) has been introduced to accurately store big integers.

### 3.3 Supported constructs

This section lists the main SystemC constructs. For each of them is specified:

- if it is recommended by OSCI to be supported by synthesis tools;
- if its is supported for synthesis by the Celoxica *Agility Compiler*;
- if it is currently supported by *SC2HIF*.

<b>SystemC Constructs</b>	<b>Recommended by OSCI</b>	<b>Synthetizable by Agility</b>	<b>Supported by SC2HIF</b>
SC_MODULE	x	x	x
SC_MODULE + template	x	x	
SC_CTOR	x	x	x
SC_HAS_PROCESS	x	x	x
inheritance	x	x	
virtual inheritance	x		

<b>SystemC Constructs</b>	<b>Recommended by OSCI</b>	<b>Synthesizeable by Agility</b>	<b>Supported by SC2HIF</b>
class nesting (with pointers)		x	
class nesting (without pointers)	x		
ports sc_in/sc_out/sc_inout	x	x	x
sc_in_clk	x	x	x
sc_in_resolved	x		x
sc_in_rv	x		x
sc_out_clock	x	x	x
sc_out_resolved	x		x
sc_out_rv	x		x
sc_inout_clock	x	x	x
sc_inout_resolved	x	x	x
sc_inout_rv (tristate port)	x	x	x
port read/write	x	x	x
positional binding			
named binding		x	
sc_signal	x	x	x
sc_signal_resolved	x		x
sc_signal_rv	x		x
sc_fifo		x	
sc_fifo operations: num_avail, num_free			
sc_thread		x	x

<b>SystemC Constructs</b>	<b>Recommended by OSCI</b>	<b>Synthesizable by Agility</b>	<b>Supported by SC2HIF</b>
sc_thread	x	x	x
sc_method	x	x	x
wait()	x	x	x
wait(n)	x (with restrictions)		x
wait_untill	x (with restrictions)		x
reset_signal_is sc_thread) (with		x	
posedge()/negedge()	x	x	x
pos()/neg()	x	x	x
sc_spawn / sc_spawn_options into a SC_FORK/SC_JOIN		x (with restrictions)	
functions of sc_spawn_options: dont_initialize, set_stack_size(N), spawn_method			
C++ types	x	x (with restrictions)	x
pointers	x (with restrictions)	x (with restrictions)	
references	x	x (with restrictions)	x
classes	x (with restrictions)	x (with restrictions)	x
virtual member functions	x	x (with restrictions)	
unions			
bitfields	x		
SystemC numeric types	x (partially and with restrictions)	x (partially and with restrictions)	x (partially)
operations on numeric types of SystemC	x (with restrictions)	x (with restrictions)	x

<b>SystemC Constructs</b>	<b>Recommended by OSCI</b>	<b>Synthetizable by Agility</b>	<b>Supported by SC2HIF</b>
operations on C++ types	x	x (with restrictions)	x
new		x (with restrictions)	
delete			
conditional statements	x (with restrictions)	x (with restrictions)	x (with restrictions)
loops	x	x	x
functions	x (with restrictions)	x (with restrictions)	x
namespace / using	x	x	
exception			
math.h library		x (with restrictions)	

### 3.4 Compiling SC2HIF

---

The following steps must be done to compile *SC2HIF*:

- Get the *HifSuite* source code and de-compress it. This will create the directory "*hif\_suite*".
- Go into the directory "*hif\_suite/src*".
- (Optional) Type "*make clean*", if it is necessary to remove all already compiled code.
- Type "*make sc2hif*".

The executable will be placed into the directory "*hif\_suite/bin*".

The following libraries and programs are required during compilation and at runtime:

- Gnome Librarys
- Java Runtime Enviroment 1.5
- LibXML2++ & LibXML2
- Apache Xerxes Library
- GCC C++ Preprocessor v. 3.3.5
- SH - The Shell Enviroment
- SED bash command
- TR bash command
- Yacc (BYacc)
- Bison

### 3.5 Running SC2HIF

---

The following steps must be done to run *SC2HIF*:

- Export the path of the directory where the *HifSuite* is installed. Such a path must be exported into an environment variable named "*HIF\_DIR*".

- Run the *SC2HIF* with the preferred parameters. To get a full description of all available parameters, run the tool with the option “*--help*”. Some basic parameters are:
  - *-f*: set a SystemC/source file for translation.
  - *-i*: set a SystemC/header file for translation.
  - *-o*: set the output HIF file name.

## 3.6 Bug list

---

### Bug 1

- **Type:** *sc2ast* bug.
- **Description:** *sc2ast* cannot parse array without elements.
- **Example:**

```
int indexTable[16] = {-1, -1, -1, -1, 2, 4, 6, 8};
```

### Bug 2

- **Type:** *sc2ast* bug.
- **Description:** *sc2ast* cannot parse a const initialization outside the class definition.
- **Example:**

```
SC_MODULE(port)
{
public:
    static const int c;
    void main() {}
    SC_CTOR(port)
    {
        SC_METHOD(main);
    }
};
const int port::c = 0;
```

### Bug 3

- **Type:** *sc2ast* bug.
- **Description:** *sc2ast* wrongly parses *sc\_int<N>* when it used as type cast.
- **Example:**

```
SC_MODULE(port)
{
public:
    sc_int<3> i;
    void main()
    {
        i = sc_int<3>(7);
    }
    SC_CTOR(port)
    {
        SC_METHOD(main);
    }
};
```

### Bug 4

- **Type:** *scxml2hif* bug.
- **Description:** *scxml2hif* cannot handle side-effects, in such cases a warning is shown and continue.

- **Example:**

```
SC_MODULE(port)
{
public:
    int v[2];
    void main()
    {
        int index=0;

        v[index+=1] = 1; // side effects are not handled
        v[index++] = 2;
    }
    SC_CTOR(port)
    {
        SC_METHOD(main);
    }
};
```

**Bug 5**

- **Type:** scxml2hif bug.
- **Description:** *sensitive << clock.pos()* is not handled.
- **Workaround:** Use for example: *sensitive\_pos << clock;*
- **Example:**

```
SC_MODULE(port)
{
public:
    sc_in_clk clock;
    void main()
    {
    }
    SC_CTOR(port)
    {
        SC_METHOD(main);
        sensitive << clock.pos(); // Sensitive pos
    }
};
```

**Bug 6**

- **Type:** sc2ast bug.
- **Description:** *sc2ast* cannot parse classes including definition of functions with template parameters.
- **Example:**

```
SC_MODULE(port)
{
public:
    sc_in_clk clock;
    template <typename T>
    void foo(T t)
    {
        int a = 0;
    }
    void main() {}
    SC_CTOR(port)
```

```
    {  
        SC_METHOD(main);  
        sensitive << clock;  
    }  
};
```

**Bug 7**

- **Type:** scxml2hif bug.
- **Description:** *scxml2hif* does not allow function to return a template type.
- **Example:**

```
template <typename T>  
T foo(T t)  
{  
    int a = 0;  
    return t;  
}
```

## 4. VHDL2HIF

---

This Section describes *VHDL2HIF*, a tool that encapsulates the *VHDL2AIF* program, developed during the Symbad project, to translate VHDL design descriptions into HIF design descriptions. The Section describes which constructs are supported by *VHDL2HIF*, how to compile and run it, and the list of bugs inherited into *VHDL2HIF* from *VHDL2AIF* that cannot be fixed since *VHDL2AIF* sources are not available.

### 4.1 Supported constructs

---

This section lists the main VHDL constructs. For each of them is specified if it is currently supported by VHDL2HIF.

<b>VHDL constructs</b>	<b>Supported by VHDL2HIF</b>
Entity	x
Architecture	x
Configuration	
Package	x (partially)
Libraries	x
Enumeration type	x
Integer type	x
Physical type	
Floating type	
Array	x
Record	x
Access/file	
Constant declaration	x
Signal declaration	x
Variable declaration	x
File declaration	
Interface/ports declaration	x
Component declaration	x
Attribute declaration	X (partially)
Wait statement	

<b>VHDL constructs</b>	<b>Supported by VHDL2HIF</b>
For loop statement	x
While loop statement	x
Process	x
Assertion	
Component instantiation	x

## 4.2 Compiling VHDL2HIF

---

To compile *VHDL2HIF* the following packages are required:

- the Java Development Kit or the Java Runtime Environment;
- the YACC (or BYACC) program ;
- the Bison program;

Moreover, the `JAVA_HOME` environment variable must contains the path where Java is installed.

After decompressing *VHDL2HIF* source files, the following steps must be done:

- enter into sub-directory "*src*"
  - > `cd src/`
- run the Makefile with "*clean*" as parameter. This step is optional.
  - > `make clean`
- run Makefile, with parameter "*hif2hdl*" to create an executable.
  - > `make vhd2hif`

## 4.3 Running VHDL2HIF

---

To run *VHDL2HIF*, the following steps must be done (*VHDL2HIF* executable name is "*vhd2hif*"):

- Export the path of the directory `$HIF_SUITE/etc/vhdl_packages`. Such a path must be exported into the environment variable "`VHDL_LIB_PATH`". `$HIF_SUITE` is the path of the *HIFSuite* installation directory.
- Run the *VHDL2HIF*, with the following parameters (respecting the order):
  - `<input.vhd>`: set the VHDL input file name.
  - `-A work.<entity>.<architecture>`: where `<entity>` is the VHDL design entity name and `<architecture>` is the VHDL design architecture name. This permits to generate a specific entity for a specific architecture as an HIF description.
  - `-o <output.rx6>`: set the HIF output file name.

## 4.4 Bug list

---

### Bug 1

- **Type:** vhd2aif bug.
- **Description:** Bitstreams are not handled. They are translated into strings.
- **Example:**

```
B"1010101011"
```

### Bug 2

- **Type:** vhdl2aif bug.
- **Description:** Unlabelled processes are named with the same label. In VHDL a process does not need a label, but when the process is translated into HIF, the label is required. The translator gives to each unnamed process the same label "process".
- **Example:**

VHDL source code:

```
process( ... )
...
end process;

process( ... )
...
end process;
```

HIF output code:

```
(STATETABLE process
...
)

(STATETABLE process
...
)
```

expected output code:

```
(STATETABLE process
...
)

(STATETABLE process1
...
)
```

### Bug 3

- **Type:** vhdl2aif bug.
- **Description:** when the initialization of a VHDL array is translated, the HIF output code misses the *AGGREGATE-ALT*-list.
- **Example:**

VHDL source code:

```
type R_Vect is array (0 to 1) of integer;
constant Rm : R_Vect := (0, 1);
```

HIF output code:

```
(TYPEDEF R_Vect (ARRAY (UPTO 0 1) (OF (INTEGER))))
(CONSTANT Rm (TYPEDEF R_Vect)(INITIALVALUE (AGGREGATE)))
```

expected output code:

```
(TYPEDEF R_Vect (ARRAY (UPTO 0 1 ) (OF (INTEGER ))))
CONSTANT Rm (TYPEDEF R_Vect ) (INITIALVALUE (AGGREGATE
(ALT 0 0) (ALT 1 1))))
```

**Bug 4**

- **Type:** vhdl2aif bug.
- **Description:** qualified expressions are not supported.
- **Example:**

VHDL source code:

```
variable v : string := string("110010");
```

**Bug 5**

- **Type:** vhdl2aif bug.
- **Description:** scalar type are not supported.
- **Example:**

VHDL source code:

```
type short is range -128 to 127;
```

expected HIF output code:

```
(TYPEDEF short (INTEGER (RANGE (TO -128 128))))
```

**Bug 6**

- **Type:** vhdl2aif bug.
- **Description:** type casting is recognized as access to an array.
- **Example:**

VHDL source code:

```
variable v : integer := integer(2);
```

output error message:

```
Unsupported VhdlIndexedName
```

expected HIF output code:

```
(VARIABLE V (INTEGER)(INITIALVALUE (CAST (INTEGER)(2))))
```

**Bug 7**

- **Type:** vhdl2aif bug.
- **Description:** only the SystemC-compatible values of *std\_logic* are supported. *std\_logic* is translated into the HIF *bit* type which allows only the four SystemC values.
- **Example:**

VHDL source code:

```
variable v : std_logic := ' - ';
```

output error message:

```
Unsupported character literal ' - '
```

#### Bug 8

- **Type:** vhdl2aif bug.
- **Description:** it is impossible to refer to a user-defined package by an extern entity.
- **Example:**

VHDL source code:

```
package pk1 is  
constant c : integer := 1;  
...  
end pk1;  
...  
variable v : integer := pk1.c;
```

output error message:

```
ERROR: pk1 is not declared
```

#### Bug 9

- **Type:** vhdl2aif bug.
- **Description:** matrix attributes with the N parameter are translated without N. The program does not output any warning or error message.
- Attribute range is translated into `__ATTR_unsupported` without printing warnings.
- **Example:**

VHDL source code:

```
type matr is array (1 to 4, 23 to 56) of bit;  
variable v : matr;  
variable N : integer;  
...  
v'length(N);
```

HIF output code (only length access):

```
(FCALL __ATTR_length  
(PARAMETERASSIGN param v)  
)
```

expected HIF output code:

```
(FCALL __ATTR_length  
(PARAMETERASSIGN param v)  
(PARAMETERASSIGN param1 N)  
)
```

## 5. HIF2HDL

*HIF2HDL* is a tool implemented by UNIVR to translate HIF descriptions into VHDL and SystemC code. It can be seen as the natural complementary tool of *VHDL2HIF* and *SC2HIF*.

### 5.1 Supported constructs

All constructs supported by *VHDL2HIF* and *SystemC2HIF* are also supported by *HIF2HDL*. Moreover, *HIF2HDL* supports the OSCI TLM constructs reported in the following table. Thus, if a tool that manipulates HIF descriptions generates TLM constructs, *HIF2HDL* will generate the corresponding TLM SystemC code. Obviously, TLM HIF constructs cannot be converted into VHDL code.

<b>SystemC TLM Constructs</b>	<b>Recommended by OSCI</b>	<b>Supported by HIF2HDL (for SystemC)</b>
sc_port	x	x
sc_export	x	x
basic_initiator_port<>	x	x
basic_request<>	x	x
basic_response<>	x	x
basic_status write()	x	x
basic_status read()	x	x
basic_slave_base	x	x
tlm_blocking_get<>	x	x
tlm_blocking_put<>	x	x
tlm_blocking_peek<>	x	x
tlm_nonblocking_get<>	x	x
tlm_nonblocking_put<>	x	x
tlm_nonblocking_peek<>	x	x
tlm_fifo<>	x	x

<b>SystemC TLM Constructs</b>	<b>Recommended by OSCI</b>	<b>Supported by HIF2HDL (for SystemC)</b>
tlm_req_rsp_channel<>	x	x
get()	x	x
put()	x	x
peek()	x	x
nb_get()	x	x
nb_put()	x	x
nb_peek()	x	x

## 5.2 Compiling HIF2HDL

---

In order to compile *HIF2HDL* the following steps must be done:

- Get the *HifSuite* source code and de-compress it. This will create the directory "*hif\_suite*".
- Go into the directory "*hif\_suite/src*".
- (Optional) Type "*make clean*", if it is necessary to remove all already compiled code.
- Type "*make hif2hdl*".

The executable will be placed into the directory "*hif\_suite/bin*".

The following libraries and programs are required during compilation and at runtime:

- Java Runtime Environment 1.5;
- GCC C++ Preprocessor v. 3.3.5;
- SH - The Shell Environment;
- Yacc (BYacc);
- Bison.

## 5.3 Running HIF2HDL

---

The following steps must be done to run *HIF2HDL*:

- Export the path of the directory where the *HIFSuite* is installed. Such a path must be exported into the environment variable "*HIF\_DIR*".
- Run *HIF2HDL* with the preferred parameters. The two possibilities are:
- *./hif2hdl -L RTL <hif\_file\_name>*: to get the VHDL code.
- *./hif2hdl -H SYSTEMC <hif\_file\_name>*: to get the SystemC code.

*<hif\_file\_name>* is a placeholder for the name of the HIF file containing the module to be translated.

Output files will be placed into the directory "*rx6out\_session*".

## 5.4 Bug list

---

No bug has been discovered up to now.

## 6. EFSM generation and manipulation via HIF

This Section describes the tool *Phase1* developed by UNIVR to manipulate EFSMs. The Section also describes how to compile and run *Phase1*.

The general architecture of *Phase1* is shown in Figure 5.

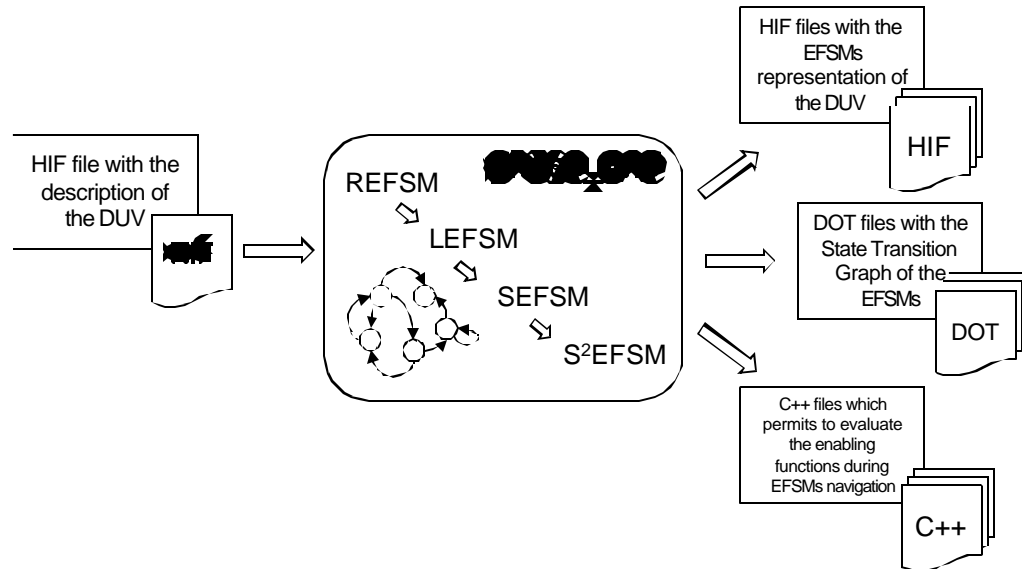


Figure 5: *Phase1* flow.

*Phase1* is a C++ program, that generates different kind of EFSMs starting from an HIF description composed of one or more processes.

*Phase1* generates the following kinds of EFSMs:

- Reference EFSM (REFSM);
- Largest EFSM (LEFSM);
- Smallest EFSM (SEFSM);
- Semi-Stabilized EFSM (S<sup>2</sup>EFSM).

The features of such different kinds of EFSMs are described in Section 4.4 of Deliverable D2.1.

*Phase1* produce the following output:

- the HIF description of REFSM, LEFSM, SEFSM and S<sup>2</sup>EFSM;
- a set of State Transition Graphs described by using the DOT language, one for each EFSM (<http://www.graphviz.org>);
- a C++ file which is used by the Laerte++ ATPG to interface with the Constraint Solver ECLiPSe (<http://eclipse.crosscoreop.com>) during the exploration of the S<sup>2</sup>EFSMs of the design under verification (DUV).

### 6.1 Compiling *Phase1*

The following steps must be done to compile *Phase1*:

- Get the *Phase1* source code and de-compress it. This will create the directory "*phase\_one*".
- Move into the directory "*phase\_one /src*".
- (Optional) Type "*make clean*", if it is required to remove all already compiled code.
- Type "*make*".

The executable of the tool will be placed into the directory "*phase\_one/bin*".

The following libraries and programs are required during compilation and at runtime:

- Boost Libraries
  - bgl-viz;
  - boost\_regex;
  - boost\_program\_option;
  - boost\_filesystem.
- *HIFSuite* library.

## 6.2 Running Phase1

---

The following steps must be done to run *Phase1* (*Phase1* executable name is "*phase\_one*"):

- Export the path of the directory where *Phase1* is installed. Such a path must be exported into the environment variable "*PHASE\_ONE\_HOME*".
- Run *Phase1* with the preferred parameters. To get a full description of all available parameters, run the tool with the option "*--help*". Some basic parameters are:
  - *-H [ --help ]*: print the help message
  - *-S [ --hif ] <arg>*: set the HIF source file name
  - *-C [ --clock ] <arg>*: set the clock signal name (the default value is "clock")
  - *-R [ --reset ] <arg>*: set the reset signal name (the default value is "reset")
  - *-V [ --state\_var ] <arg>*: set the state variable name
  - *-M [ --model ] <refsm | lefsm | sefsm | ssefsm | h added>*: specify which computational model has to be generated

## 6.3 Bug list

---

### Bug 1

- **Type:** phase1 bug.
- **Description:** In some case, State Transition Graph generated by *Phase1* does not correspond to the real EFSM topology. The problem leads to an internal data structure used to memorize EFSM transitions and the related iteration operator. The data structure is a map, the transition identifiers are keys of the map and the identifier is of type string. The internal ordering function of the map uses lexicographic ordering. Instead, the expected extraction ordering should be the insertion ordering (like a FIFO list). This erroneous assumption reflects into a wrong STG generation.
- **Example:** Consider 40 transitions outgoing from state s0 of an EFSM.

Expected ordering:

```

transition_id_s0_#1
transition_id_s0_#2
transition_id_s0_#3
transition_id_s0_#4
transition_id_s0_#5
[...]
transition_id_s0_#9
transition_id_s0_#10
transition_id_s0_#11
[...]
transition_id_s0_#19
transition_id_s0_#20
transition_id_s0_#21
[...]
transition_id_s0_#29
transition_id_s0_#30

```

```
transition_id_s0_#31  
[...]  
transition_id_s0_#39  
transition_id_s0_#40
```

Ordering generated by Phase 1 based on the lexicographic order:

```
transition_id_s0_#1  
transition_id_s0_#10  
transition_id_s0_#11  
[...]  
transition_id_s0_#19  
transition_id_s0_#2  
transition_id_s0_#20  
transition_id_s0_#21  
[...]  
transition_id_s0_#29  
transition_id_s0_#3  
transition_id_s0_#30  
transition_id_s0_#31  
[...]  
transition_id_s0_#39  
transition_id_s0_#4  
transition_id_s0_#40  
transition_id_s0_#5  
[...]  
transition_id_s0_#9
```

## 7. HLDD generation and manipulation via HIF

This Section describes the tool developed by TUT and UNIVR to generate the set of HLDDs corresponding to the HDL description of the DUV. The HLDD paradigm is adopted by the ATPG *Decider* developed by TUT.

### 7.1 HLDD generation

An HLDD can be generated starting from the corresponding LEFSM since:

- in both the HLDD and LEFSM formats, design constraints are broken into separate expressions;
- it is easy to convert the enabling function of an EFSM into HLDD constraints;
- a path on the STG of an EFSM corresponds to a path on the tree of the corresponding HLDD and vice-versa

Thus, a sub-module of *Phase1* permits to create HLDDs from the design description by exploiting the EFSM generation flow as reported in Figure 6.

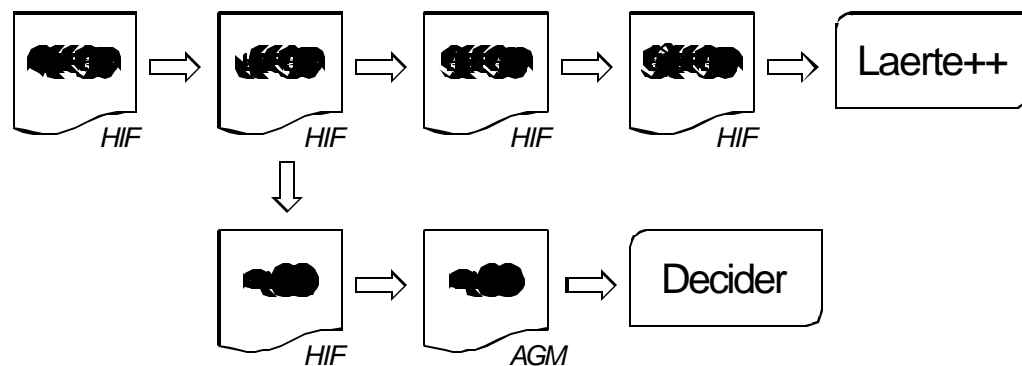


Figure 6: HLDD generation flow.

The generation of HLDDs corresponding to EFSMs allows us to integrate *Decider* and *Laerte++* to improve test pattern generation [3].

#### 7.1.1 Running Phase1 to generate HLDD

The following steps must be done to run the *Phase1* for generating HLDDs starting from an HIF description:

- Export the path of the directory where *Phase1* is installed. Such a path must be exported into the environment variable "PHASE\_ONE\_HOME".
- Run *Phase1* with the preferred parameters. To get a full description of all available parameters, run the tool with the option "--help". Some basic parameters are:
  - *-H* [--help ]: print the help message
  - *-S* [--hif ] <arg>: set the HIF source file name
  - *-C* [--clock ] <arg>: set the clock signal name (the default value is "clock")
  - *-R* [--reset ] <arg>: set the reset signal name (the default value is "reset")
  - *-V* [--state\_var ] <arg>: set the state variable name
  - *-M* [--model ] *hldd*: specify HLDD computational model has to be generated

## 7.2 HLDD translation into AGM format

The EFSM (HIF format) into HLDD conversion developed by UNIVR is supplemented by the back-end tool *HLDD2AGM* for translating the HLDD diagrams into AGM format supported by the TUT verification tools. The task of the translator includes parsing the HLDD structures created by *Phase1*, converting the control constructs of the HLDD files into non-terminal nodes and the data assignments into terminal nodes in the resulting AGM file.

The back-end translator *HLDD2AGM* has been coded in Java. It has a graphical user interface for opening HLDD files and for specifying the output location of the AGM files (see Figure 7).

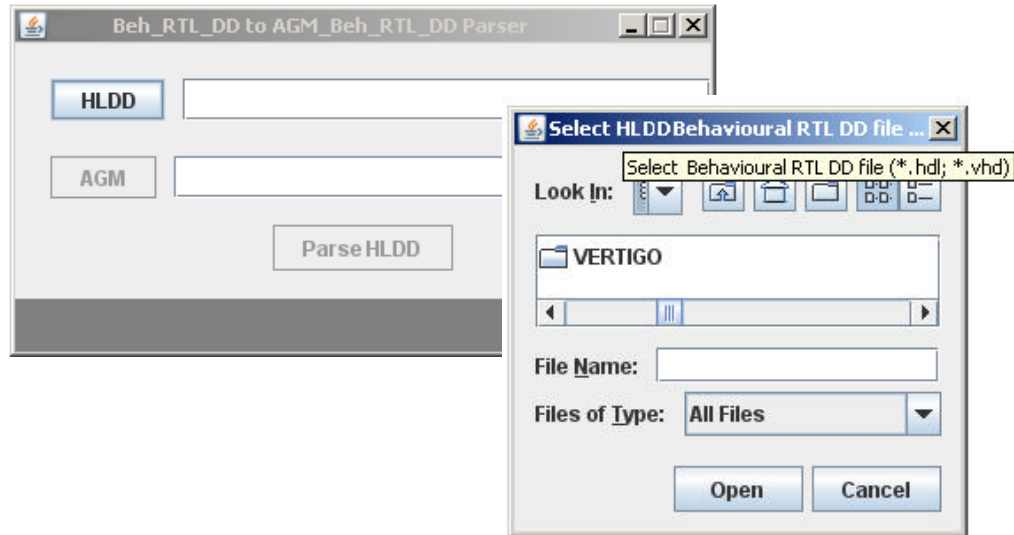


Figure 7: HLDD generation flow.

### 7.2.1 Installing the HLDD2AGM tool

*HLDD2AGM* tool is provided with a single installation file. The destination directory is selected by the user during the installation.

The only requirement to run *HLDD2AGM* tool is to have Java Runtime Environment (at least version 1.5) installed. This can be downloaded from Java Sun web site (<http://java.sun.com>).

### 7.2.2 Running HLDD2AGM translation

In order to run *HLDD2AGM*, the user simply has to either select *HLDD2AGM* from the *Start* menu under Windows environment, or run `./hldd2agm` from the installing directory under Unix environment. *HLDD2AGM* user interface allows user to select an input HLDD file with a file browser and to define an output AGM file (default location and name are the same as of the HLDD file).

### 7.2.3 Bug list

The translator supports machine-generated code obtained from *Phase1* and there are no known bugs identified.

## 8. PRES+ generation and manipulation via HIF

---

This section introduces a tool for automatic generation of PRES+ (Petri-net based Representation of Embedded Systems) models given an HIF representation of the system. This tool is called *hif2pres*. In the VERTIGO flow, PRES+ models are mainly used for formal verification (model checking). As such, additional semantic information is needed to formally describe the intended concurrency (scheduling of processes) and communication mechanism. That information is slightly different between, for instance, SystemC, VHDL or a user specific semantics. Figure 8 provides an overview of this procedure. As indicated in the figure the semantic information is provided in a C++ library.

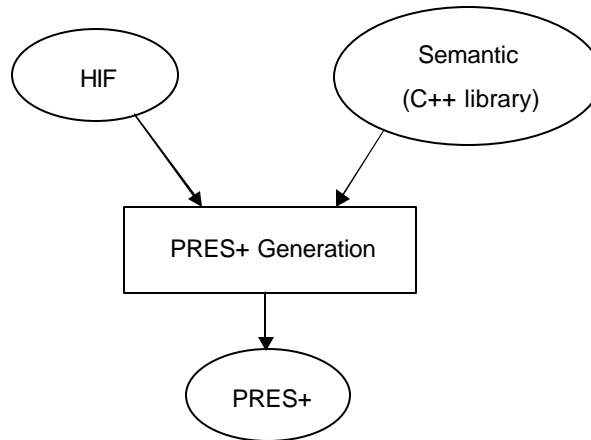


Figure 8: PRES+ generation overview

The use of PRES+ provides an efficient and convenient means of capturing high-level and real-time aspects of a system. Traditionally, in the context of hardware design, formal methods are applied on low-level RTL designs. PRES+, however, allows for applying these methods on high-level TL designs. Models containing real-time time bounds, high-level communication channels, dynamic loop bounds, events etc. can be formally verified via PRES+. These features are (or will be) only implemented to the extent supported by HIF.

### 8.1 Translation overview

---

The statements within a process are considered to be executed sequentially, whereas processes are executed concurrently in the way defined by the semantics. A standard translation is therefore made for constructs related to the contents of a process, but with a standard interface towards other processes and entities. A scheduler, defined by the particular semantics used, orchestrates and coordinates the processes and signals. Figure 9 illustrates these concepts.

Each process has two ports through which it exchanges execution information: trigger and yield. When the scheduler puts a token in a trigger port, the corresponding process is allowed to execute. Once finished, the process shall put a token in the yield port in order to indicate its completion to the scheduler. The scheduler may give back execution control at a later time [4].

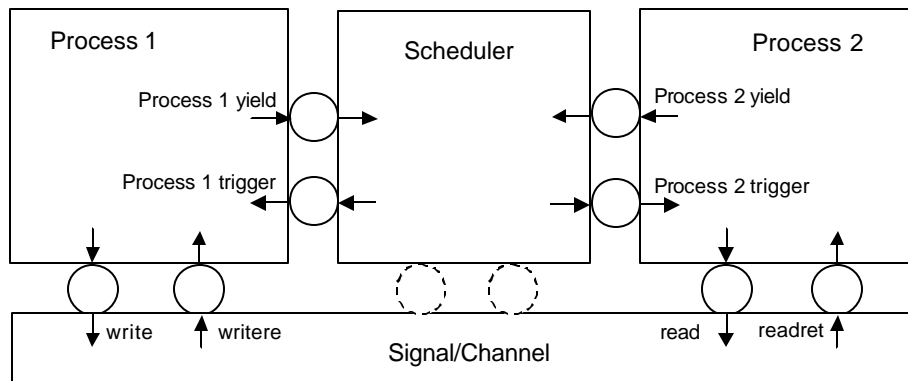


Figure 9: Standard PRES+ interfaces of HIF constructs

A port in HIF is translated into two ports in PRES+, in order to maintain sequentiality of the processes. HIF out ports translate to the PRES+ ports *write* and *writere*, and HIF in ports to *read* and *readret*. Once a process has issued a request through a *write* or *read* port, it will wait until that request has been completed, which is indicated by a token in the corresponding ret port.

Ports are, moreover, mapped to a signal or channel that models the inter-process communication. Depending on the particular semantics used, the signal (or channel) may also need to interact with the scheduler. This is in particular necessary when using a cycle-based semantics, and where signals need to perform an update at every new cycle.

## 8.2 Supported constructs

The following constructs are currently supported by the tool we have implemented:

- single instance processes, signals, sensitivity on signals;
- variables of integer type, constants, typedefs (enums);
- assignments to variables, ports and signals, arithmetic expressions;
- case, switch, while loops ;
- wait and suspend for a constant amount of time (expressed in the default time unit);
- function definitions, function calls (maximum one process simultaneously executing a function);
- SystemC scheduling and communication semantics.

## 8.3 Compiling HIF2PRES

The core part of *hif2pres* is implemented in the class *HifToPresTranslator*. This makes the PRES+ generation algorithm very modularised and easy to integrate as part of a larger tool. It moreover consists of three other classes, *SimulationSemantic*, *SystemCSimulationSemantic* and *NoSimulationSemantic*, where the latter two inherit the first. *SimulationSemantic* is consequently an abstract class from which developers should inherit in order to implement a customised simulation semantic. The class *SystemCSimulationSemantic* implements the semantic for SystemC, whereas the class *NoSimulationSemantic* implements an empty semantic without concrete contents.

Apart from including and linking to the HIF library, as described in Section 10, the same must also be done to the PRES+ library. The PRES+ library consists of two files: *libbasics.so* and *libextensions.so*. Both files must be linked with *hif2pres*.

The library moreover makes use of *xerces* for parsing XML descriptions of PRES+ models. Therefore, it is also necessary to link this library with the tool.

## 8.4 Running HIF2PRES

---

In order to run the *hif2pres* tool, it is necessary to add the directory where the PRES+ library files can be found to the environment variable "*LD\_LIBRARY\_PATH*". On Unix based systems this can be done with the following command:

```
> export LD_LIBRARY_PATH=PRES+directoryhere:$LD_LIBRARY_PATH
```

The tool is executed as described below:

```
> hif2pres [-sc] source.rx6 target.xml
```

The flag *-sc* indicates that SystemC semantics should be applied on the generated PRES+ model. This is currently the only supported semantic. If the flag is left out, no semantic specific entities will be generated.

## 9. Property management via HIF

The property management feature of *HIFSuite* aims at joining Assertion-based Verification (ABV) with dynamic verification based on ATPG. Temporal properties expressed in PSL are firstly translated into SystemC modules (called checkers) that allow us to dynamically check the satisfiability of the properties, and then integrated with an ATPG (e.g., *Laerte++*) by means of a proper framework, as described in the next subsections. Figure 10 shows the step sequence of the automatic translation. Checkers can be used also by industrial tools like *imPROVE-HDL* and *Assertains*.

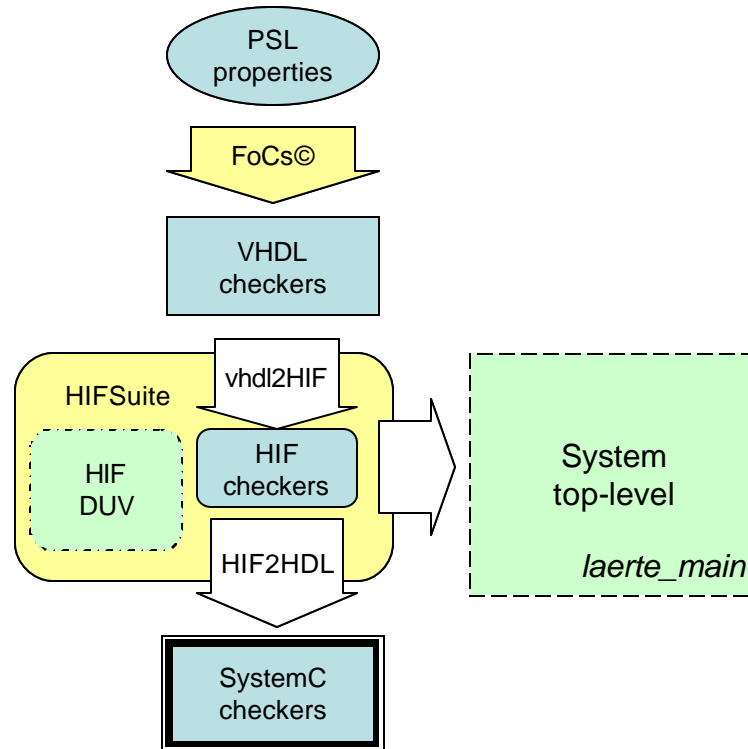


Figure 10: Checker generation flow

PSL properties are translated into VHDL checkers by using *IBM FoCs©*. Then, *VHDL2HIF* and *HIF2HDL* are used to translate the VHDL checkers into SystemC code, moving through the HIF intermediate format. During this double step, a sub-module of *HIFSuite* analyses the HIF description provided by *VHDL2HIF* in order to automatically generate the top level module (called *laerte\_main*) suitable to correctly link the checkers and the DUV to *Laerte++*. The *laerte\_main* framework is shown in Figure 11.

*Laerte++* generates stimuli for the DUV, while the checker module monitors both the inputs and output ports of the DUV. At every clock cycle, the checkers notify the ATPG about the properties status by means of a proper signal (i.e., *prop\_failure*).

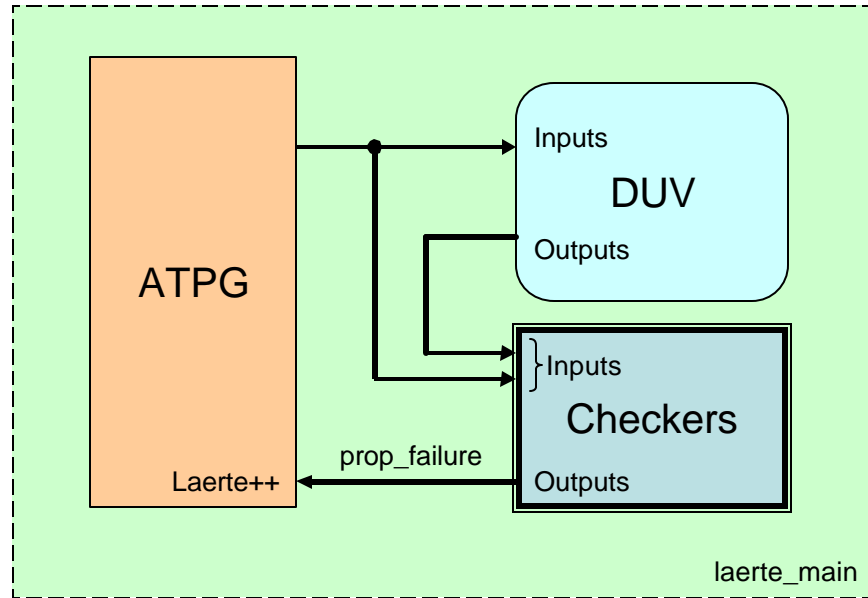


Figure 11: Structure of the *laerte\_main* to connect DUV, checkers and ATPG.

## 9.1 Setting up and running FoCs

*IBM FoCs* automatically translates temporal properties into checkers. Properties can be expressed both in Sugar and PSL (PSL flavour GDL or PSL flavour Verilog). The quick setting of *FoCs* consists of 3 steps:

1. Set the target language of the checkers to be generated (VHDL, Verilog, C++). In our flow we exploit *FoCs* to generate VHDL checkers, since VHDL is adopted as input format also by *imPROVE-HDL* and *Assertains*.
2. Set the clock name and the clock polarity (rising edge, falling edge or both).
3. Set the reset name and reset polarity.

The tool is ready to run once the compulsory information listed above have been filled out. However, more accurate choices can be done, by setting parameters like language subset (i.e., VHDL-93 synthesizable or full), signal datatype, reset mode, optimization level, etc.

Each property is translated into one checker (called *vunit*) and all the properties are held into a single VHDL file, provided with a library package for data types management (i.e., *sim\_support\_downto*).

## 9.2 VHDL checkers

VHDL checkers consists of a VHDL entity and one or more processes which implement the correspondent property and that are sensitive to the clock signal. At every clock cycle, the checker processes get the values of the input ports. A property failure corresponds to an assertion failure with a report information in the VHDL code. In the property management chain, this mechanism is converted by *HIFSuite* into a signal setting, which reports the property status (i.e., property holds, property fails), as explained in the next Section.

The interface has one input port for each signal or variable used in the PSL property formulation. An example is the following:

```
library ieee;
```

```

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.sim_support_downto.all;
ENTITY root_plus IS
  PORT (
    clk           :IN std_logic;
    reset         :IN std_logic;
    number_port   :IN std_logic;
    number_isready :IN std_logic;
    result_isready :IN std_logic;
    result_port   :IN std_logic);
END root_plus ;

ARCHITECTURE checker OF root_plus IS
  ...

```

Entity `root_plus` represents the checker of the following property:

```

vunit root_plus
{
  assert
  always ((number_port>0 & number_isready=1) ->
  ((result_isready=0) until (result_isready=1 &
  result_port>0)));
}

```

The VHDL checkers and the package of datatype management generated by *FoCs* are the inputs for *VHDL2HIF*. Currently, all the constructs used by *FoCs* to implement checkers are supported by *VHDL2HIF* excepting the following:

- *wait until*
- *numeric\_std library*

One bug of *FoCs* has been caught out and signalled to the IBM developers. It concerns the process name in case of multi-process entity.

## 9.3 Checker conversion via HIFSuite

The checker conversion task via *HIFSuite* consists of three steps. First the VHDL code generated by *FoCs* is translated into HIF format by means of *VHDL2HIF*. For example, the HIF code corresponding to the *root\_plus* property is the following:

```

(DSIGNUNIT root_plus
  (VIEW checker
    (VIEWTYPE " ")
    (DESIGN HARDWARE)
    (LIBRARY sim_support_downto Aif " ")
    (INTERFACE
      (PORT clk (IN )(BIT (RESOLVED)))
      (PORT reset (IN )(BIT (RESOLVED)))
      (PORT number_port (IN )(BIT (RESOLVED)))
      (PORT number_isready (IN )(BIT (RESOLVED)))
      (PORT result_isready (IN )(BIT (RESOLVED)))
      (PORT result_port (IN )(BIT (RESOLVED)))
    )
    (CONTENTS
      ...

```

Then, a submodule of *HIFSuite* is run to add one output port to the checker interface, which signals whether the property fails or not. This step is needed since the checker will be linked to the ATPG for the dynamic simulation. Thus, the checker will inform about the property status with a signal value rather than stopping the simulation in case of property failure. The code generated by the tool corresponding to the *root\_plus* property is the following:

```
(DESIGNUNIT root_plus
  (VIEW checker
    (VIEWTYPE "")
    (DESIGN HARDWARE)
    (LIBRARY sim_support_downto Aif "")
    (INTERFACE
      (PORT clk (IN )(BIT (RESOLVED)))
      (PORT reset (IN )(BIT (RESOLVED)))
      (PORT number_port (IN )(BIT (RESOLVED)))
      (PORT number_isready (IN )(BIT (RESOLVED)))
      (PORT result_isready (IN )(BIT (RESOLVED)))
      (PORT result_port (IN )(BIT (RESOLVED)))
    )
    (PORT prop_failure (OUT )(BIT (RESOLVED)))
  )
  (CONTENTS
    ...
  )
)
```

Finally, the modified HIF code is translated into SystemC by means of *HIF2HDL*. Currently, all the HIF constructs used to express checkers are supported by *HIF2HDL*.

Besides generating SystemC checkers, which is explained in the next section, *HIFSuite* analyses the syntactical structure of checkers as well as the DUV in order to automatically generate the top-level module (i.e., *laerte\_main*) which instances and links all the components for the simulation.

## 9.4 SystemC checkers

---

The SystemC code representing the checker is automatically generated starting from the HIF checker, by using *HIF2HDL*. The generated SystemC checker is semantically equivalent to the HIF checker. By referring to the SystemC syntax, the checker is implemented with a *SC\_MODULE* in which *SC\_METHOD* processes wake up at every clock cycle. Inputs values are monitored in order to set the output port corresponding to the property status. For example, the SystemC code of the *root\_plus* property is the following:

```
SC_MODULE(root_plus) {
public:
  sc_in    <sc_logic > clk;
  sc_in    <sc_logic > reset;
  sc_in    <sc_logic > number_port;
  sc_in    <sc_logic > number_isready;
  sc_in    <sc_logic > result_isready;
  sc_in    <sc_logic > result_port;
  sc_out   <sc_logic > prop_failure;
  void process();

  SC_HAS_PROCESS(root_plus);
  root_plus(sc_module_name name_) : sc_module(name_)

{
```

```
        SC_METHOD(process);  
        sensitive << clk;  
    }  
    ...
```

Currently, all the HIF constructs used to implement checkers are supported by *HIF2HDL*.

## 10. A mini hackers guide for HIF tools

---

The structure of HIF-based tools is quite complex, thus, this Section is intended as a guide for everyone wants to extend the functionalities of *HIFSuite*.

### 10.1 General tips

---

- Check the documentation placed into the directory `"/doc"`.
- Re-build the DoxyGen documentation according with your needs. The default Doxyfile must be used to generate public documentation (i.e., its target is to allow third-parts applications development), but it is quite useless for who want to modify or extend the *HIFSuite* source code.
- Read the makefiles to get information about which files are dynamically built during compilation.

### 10.2 The RxApp library

---

*RxApp* is the library upon which all the rest of *HIFSuite* is build. Its scope is to allow easy configuration loading, independently from the platform.

To work properly, this library uses a main configuration file, called *hif\_config.txt*, whose location path must be exported before run the code. Thus, all tools in *HIFSuite* need to export the *HIF\_DIR* variable. For instance, if the configuration file is placed into `"/etc"`, and the *HIF2HDL* executable is included in the global path, the following export is required to run *HIF2HDL*:

```
> export HIF_DIR=/.
```

```
> hif2hdl
```

The *RxApp* library is used also to load other subprograms or libraries, like in the following code:

```
#include <rxapp.h>
int main() {
// Read the standard config file.
RxCfg.load(FileStructure::eval("${HIF_DIR}$SEP$etc$SEP$hif_config.txt"));
// Read the user config file.
RxCfg.load(FileStructure::eval("${HOME}$SEP$.hif$SEP$hif_config.txt"), 1);
// Find the base path. It is supposed that a variable
// called HIF_EXAMPLE is defined in hif_config.txt
baseDir=RxCfg["HIF_EXAMPLE"]+FileStructure::eval("$SEP");
// now it is possible to load all the files placed into the "HIF_EXAMPLE"
// directory.
... ..
}
```

The function *FileStructure::eval()* takes a string, and is substitutes into it all occurrences of the marker `$SEP$` with the path separator. Thus, the function *FileStructure::eval()* permits to create portable code.

To use the *RxApp* library, simply include the header *rxapp.h* and link the code with *librxapp.a*.

### 10.3 The HIF library

The HIF library defines all HIF objects and a lot of useful features to manipulate them.

The most important subdirectories of *hif/* are:

- *include/hif/classes*: which contains a header file for each of the HIF objects.
- *src/zlib*: which contains the C source code of a compression library. This is used because HIF can be described into two formats: plain text (.rx6), or binary code (.bif3), which should be a compressed gzipped file.
- *src/bif*: which contains a library which parses the input/output data to be passed to *zlib*.
- *src/semantics*: which contains some functions for consistency checking, and allows HIF tree manipulation based on the language semantics (e.g., simplification of expressions).
- *src/utills*: which contains some useful functions to manipulate an HIF object tree.
- *src/visitors*: which contains visitors for traversing the HIF object tree in a recursive way. Each HIF object has an *accept\_visitor()* method, which takes a visitor object and calls a visiting method specific for the target HIF object type.

Other useful functions included in the HIF library are:

- *write\_aif\_file(const char \* file, Object \* obj)* : which writes into a file the HIF tree whose root is *obj*. The type of output (rx6 or bif3) is auto-detected from the file extension.
- *Object \* read\_aif\_file(const char \* filename)*: which loads an HIF tree from a file. The input file is parsed according to its extension.

## 10.3.1 HOWTO extend the HIF Library - Guidelines

The following steps must be done to extend the HIF Library:

1. Check the HIF object tree (rooted by the Object class), to decide the best position to place the new HIF object. Please, remember that the HIF data structure is a tree, hence only a single inheritance is allowed.
2. Re-implement for the new HIF object all the inherited virtual methods and implement the new ones.
3. Write a DoxyGen-parserizable documentation (it is suggested to use the three-slashes comment style.) for:
  - not overridden methods,
  - overridden members that requires a more detailed documentation;
  - new members variables;
  - the class itself.
4. Write documentation for at least the main complex algorithms, with a not-DoxyGen-parserizable comment. It is suggested to use the two-slashes comment style.
5. Extend all the HIF classes and methods related to the new object, like the HIF visitors.
6. Debug the new features (documentation included) and check that all the HIF features are supported with the new ones.
7. If needed, extend the HIF front-end and back-end tools, like HIF2HDL, SC2HIF, etc..

## 10.4 HIF2HDL

---

*HIF2HDL* sources are divided into three subdirectories:

- *hif2hdl/backend\_commons*: which contains features of *HIF2HDL* common to both VHDL and SystemC, like its initialization, the initial configurations, etc.
- *hif2hdl/backend\_sysmodel*: which contains code specific for the HIF to SystemC translation.
- *hif2hdl/backend\_hardware*: which contains code specific for the HIF to VHDL translation.

## 10.4.1 HOWTO extend HIF2HDL - Guidelines

*HIF2HDL* must be extended each time new features/object are added to the HIF library. The following steps represent a guideline for *HIF2HDL* extension:

1. Check which output language (SystemC, VHDL or both) can support the new feature added to the HIF library.
  - a. If one of the output languages cannot support the new feature:
    - i. Try to transform it into something syntactically allowed by the language and semantically equivalent to the desired feature.
    - ii. If not possible, include code in HIF2HDL for signalling the user that the construct is not supported during the HIF2HDL conversion with the selected language and stop the translation. Writing a warning message without stopping the translation could produce semantic errors in the translated code.
2. Extend visitors and managers for visiting the HIF tree, if needed (e.g., *MakefileWriter* collection, *GeneratorBase* collection and the *C\_ParseLine* class).
3. Create a DoxyGen-parserizable documentation. Please, use the three-slashes comment style.
4. Include comments in the code for main algorithms, by using a not-DoxyGen-parserizable style. Please, use the two-slashes comment style.
5. Debug all the new features (documentation included).

## 11. References

---

- [1] M. Bombana, and F. Bruschi, "SystemC-VHDL co-simulation and synthesis in the HW domain", In Proc. of IEEE DATE, pp. 106-111, 2003.
- [2] C. Cote, and Z. Zilic, "Automated SystemC to VHDL translation in hardware/software codesign", In Proc. of IEEE ICECS, pp. 717-720, 2002.
- [3] G. Di Guglielmo, F. Fummi, M. Jenihhin, G. Pravadelli, J. Raik, R. Ubar, "On the Combined Use of HLDDs and EFSMs for Functional ATPG", Submitted to IEEE EWDTS 2007.
- [4] D. Karlsson, P. Eles, Z. Peng, "Formal Verification of SystemC Designs Using a Petri-Net based Representation", In Proc. of IEEE DATE, pp. 1228-1233, 2006.