SCAM
System**C** **A**bstract **M**odel

Tobias Ludwig

June 19, 2018

Contents

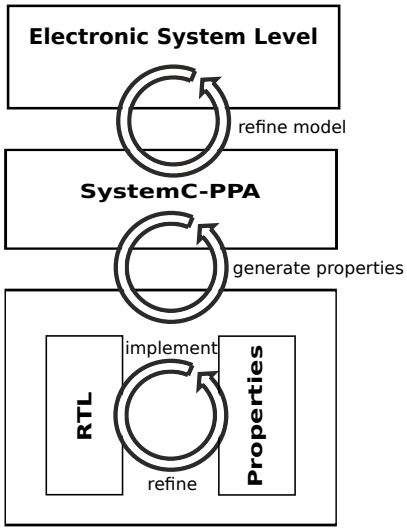
1 Introduction

In this manual a new hardware design methodology, called *Property-Driven Design* (PDD) is introduced. It borrows ideas from a software design paradigm called *Test-Driven Development* (TDD). The core idea of TDD is that the tests should guide the software engineer through the design process with the result being that any implemented code is covered by a test. The flow for designing a new feature with TDD is:

1. Create tests that describe the behavior of the feature.
2. Choose a test and implement code that fulfills the test.
3. Check if all previously checked tests are still valid.
4. Return to 2 until all tests hold on the design.

The software design process is finished if all tests hold for the design. This approach allows an incremental software design process and results in less design-bugs, a complete test coverage and a documentation of the code.

Figure 1: Overview of the desired flow



The PDD flow starts with a verified description of the component to develop at the *Electronic System Level* (ESL). From this description the tests, in form of IPC properties (e.g., SVA, ITL), are generated automatically by our tool *SCAM* ("**S**ystem**C** **A**bstract **M**odel"). Now, the hardware designer implements the hardware, as in TDD, by choosing a property and implementing the hardware that fulfills the property. Once all properties hold on the design the hardware design process is finished.

The design entry point of the PDD is the ESL. Figure 1 shows an overview of the desired flow that the flow consist of three major steps. The first step is analyzing ESL-description with *SCAM* and refactoring it according to the provided feedback. The resulting model is now considered the golden reference for the subsequent steps. The second step is the generation of abstract properties from this model with *SCAM*. Lastly, the hardware is implemented by iteratively implementing the hardware property by property. An iteration step involves implementing the hardware, refining the properties and proving the properties with a property checker (e.g., OneSpin, JasperGold, Questa).

One novelty of PDD is that the generated properties ensure that the RTL design to develop has the same I/O behavior as the ESL design. Thereby, the ESL is a sound abstraction of the RTL design. This enables using the ESL design as the golden reference and thus verification results from the system level translate to RTL. Also, any design bug from the system level will translate to the RTL. This is why we emphasize exhaustive system-level verification before starting the RTL design process.

Before we start to explain the PDD in more detail you are provided with a Walk-through in Section 2, which explains the steps of the PDD for a simple example. The *installation* of our tool is explained in Section 5.

2 Walk-Through

2.1 Step 1: ESL analysis und refactoring

The PDD-methodology starts with a system-level design, that acts as golden reference for the following design process. Here, SystemC is used as system-level design entry language. A system is composed of a se of communication modules. On system level the main focus is on describing and verifying the communication between the modules. Each of these modules is later transformed into a set of properties.

Listing 1 shows the description of such a module. The code shows a SystemC module, named *Example*, with one input and one output. The inputs and outputs are connected with other modules via channels. In this module the inputs (*b_in*) and outputs (*b_out*) use a *blocking* interface. Blocking in this context means, that underlying communication protocol is that of a four-phase handshake and the message is transmitted if and only if both sides are ready for communication.

```
1  struct Example: public sc_module{
2    //Constructor
3    Example(sc_module_name name):
4            value(9){SC_THREAD(fsm);}
```

```
 5     SC_HAS_PROCESS(Example);
 6
 7     //Ports
 8     blocking_in<int> b_in;
 9     blocking_out<bool> b_out;
10
11     //Variableless
12     int value;
13
14     //FSM
15     void fsm(){
16     while(true){
17           b_in->read(value);
18           if(value > 10){
19                 b_out->write(true);
20           }else b_out->write(false);
21     }}
22   };
```

Listing 1: Example of a SystemC module

The behavior of the module is described within the method *fsm*, that is registered as a thread to the SystemC scheduler and is executed infinitely often. The module reads a value from the input *b_in* and stores the result in the variable *value*. If *value* is >10 the output port sends *true*, *false* otherwise.
In order to describe the modules the designer is not allowed to use an arbitrary SystemC description. The reason is that not everything that is possible with SystemC is transferable to hardware, e.g., dynamic memory allocation. This is why the designer is restricted to a design subset called SystemC-PPA (designable != synthesizable). Thus, after verifying the behavior of the SystemC module, the designer has to check whether any constraints of the subset are violated by invoking SCAM with *./SCAM path-to-file -AML*. The following example shows the output of SCAM without any errors:

```
############################
Module: Example
############################

======================
PPA generation:
----------------------
[...] Metrics of PPA generation

======================
Instances
----------------------
[...] Connection between modules
[...] No connections for single modules

======================
AML: Example
----------------------
[...] Pseudo-representation of the module
```

The pseudo-code underneath *"AML: Example"* shows an abstract representation of the module. The language used here is called *"Abstract Model Language"* and is based on the specification described in [1]. A parser for this language is available, too.
The file *WalkThrough_with_error.h* contains a statement that is not part of the subset and the output of *SCAM* reports the error. If a statement is found to be not part of the subset it is ignored for the abstract model.

```
======================
Errors: Translation of Stmts for module Example
----------------------
- test.push_back(value)
-E- push_back() is not a supported method!
```

Using a std::vector on system level is beneficial when it comes to executing the simulation on the CPU but it's not transferable to a hardware system, because memory implementations have a static and known size. Thus, the statement is reported by SCAM as not being part of the subset. The designer has now to decide: If the statement is important for the module behavior, the SystemC description has to be adapted, otherwise the statement has no effect on the behavior (e.g., a std::cout) and thus the error may be ignored.

2.2 Step 2: Property generation

The second step is to generate the properties with our tool *SCAM*. The input to the tool is a single SystemC module or a system of SystemC modules. The tool parses the files and generates abstract model, that is used for the generation of the properties. In order to generate the properties run *./SCAM path-to-file/WalkThrough.h -ITL*. The *-ITL* command invokes the ITL-property generation.

```
======================
ITL: Example
----------------------
State-map(unoptimized):8 operations created
State-map(optimized):8 operations created
----------------------
[...]
```

Here, we will provide a quick overview over the generated abstract properties, for more details we refer to following sections. In order to link the abstract objects of the system-level to a concrete RTL implementation we introduce macros. The RT designer replaces the *MACRO_BODY* with concrete information from the RT design. For *b_in* and *b_out* the following macros are generated:

```
-- SYNC AND NOTIFY SIGNALS
macro b_in_notify :boolean:= MACRO_BODY end macro;
macro b_in_sync   :boolean:= MACRO_BODY end macro;
macro b_out_notify:boolean:= MACRO_BODY end macro;
macro b_out_sync  :boolean:= MACRO_BODY end macro;
-- DP SIGNALS --
macro b_in_sig : int := %BODY% end macro;
macro b_out_sig : bool := %BODY% end macro;
```

The sync and notify signals are of type boolean and are necessary for implementing the four-phase handshake and the actual message is specified as datapath macro. The notify signals evaluates to true, if the module is ready for communication and the sync signal evaluates to true if the corresponding communication partner is ready. If both sides are ready for communication the message is exchanged.
The most important step is refining the abstract representation of the system-level behavior of the module. In our methodology the generated properties capture the I/O behavior of the system-level design. The system-level thread transformed into a special FSM called PPA, with states being the communication points and transitions between states formalized as properties.
Listing 1 contains three communication calls (line 17, 19, 20). For each of those points a state is introduced, the states are called important states because they represent important control points of the hardware. The hardware designer has to specify which hardware state represents this important state. In order to do so it's allowed to use internal signals as well as outputs.

```
-- STATES --
macro run_0 : boolean :=  end macro;
macro run_1 : boolean :=  end macro;
macro run_2 : boolean :=  end macro;
```

The properties describing the transition between important states are written as IPC properties. Each property starts and ends in an important state and there is a property for each possible execution path between two communication calls. The following property starts in the important state *run_0*, that is the state after reset and in this state the design waits for a new input and describes the path for $b\_in\_sig \geq 11$. If the input is available ($b\_in\_sync == true$) and the input value is greater than 10 the hardware transitions to important state *run_1*. In *run_1* the value of *b_out_sig* is set to true and the counterpart is notified by raising *b_out_notify*. The hardware remains in state *run_1* until the counterpart accepts the handshake and the message is passed.

```
property run_0_read_0 is
dependencies: no_reset;
for timepoints:
  t_end = t+1; -- CHANGE HERE
assume:
  at t: run_0;
  at t: (b_in_sig >= 11);
  at t: b_in_sync;
```

```
prove:
  at t_end: run_1;
  at t_end: b_out_sig=true;
  during[t+1,t_end]: b_in_notify=false;
  during[t+1,t_end-1]: b_out_notify=false;
  at t_end: b_out_notify = true;
end property;
```

In case the module has to wait for the communication partner a wait-property is generated automatically for each blocking communication that enforces the hardware to remain in the same important state until the respective *sync* signal is active.

2.3 Step 3: Hardware implementation

The RT-designer has the task to specify how the abstract system level objects are implemented by the RT-design by filling out the macros. For example, the macro *b_in_sig* represents the incoming value. The designer has an infinite amount of possibilities for filling this macro. The incoming message may be transported by the input signal value_in and thus the macro is refined into

```
macro b_in_sig : int :=
  RT_design/value_in
end macro;
```

It is also possible that message is transported by two different RT signals and the resulting value is the sum of those two signals. The designer provides this information in the macro body.

```
macro b_in_sig : int :=
  RT_design/input1 + RT_design/input2
end macro;
```

In case there exist no previous implementation the RT-designer has the option to generate a VHDL-template by running *./SCAM path-to-file/WalkThrough.h -VHDL*. This template contains a package with all used datatypes and a barbone VHDL implementation matching the properties. The template is not a synthesized design and thus doesn't contain any behavioral components. A full list of available commands is provided by running *SCAM* without parameters.
In order to start the hardware design process, load your VHDL template or empty design and the accompanying properties with the property checker. The first property to prove is the reset property. The design process continues with implementing all operations starting in the important state after reset. The RT-designer has also the task to refine the abstract state macros during the design process.

3 System level and SystemC-PPA

The design flow starts at the Electronic System Level (ESL) and the system-level model is described as an executable system model, that is composed of communicating modules. Communication between modules is modeled on the transaction level so that the behavior of the modules is described as an untimed, bit-abstract model. Each module is modeled as a path predicate abstraction (PPA)[1] [2] [3] which describes the behavior as a FSM in terms of its I/O states and transitions. The behavior of the system level results from the asynchronous product of the module's behavior.
Due to the untimed behavior of the system-level model, each module may run at a different speed. In order to exchange a message between two modules they need to synchronize through a handshake, which ensures that a message is not lost during transaction. At system level this handshake is implemented through events, during the RTL design-process the handshake is realized by a four-phase handshake.
Sometimes the full handshaking bears unnecessary overhead, e.g., the designer accepts that a message may be lost. This is why we introduced three different interfaces for the ports that provide, in our experience, enough flexibility to describe any desired hardware behavior. It's the system designer's task to choose the communication scheme of a port during the system-design process.
Each interface will generate a different kind of property suite and thereby affects the hardware design process. The basic interface is called *blocking* and it implements the blocking-message passing handshake, it ensures that a message is never lost. *MasterSlave* is a special case of the blocking interface for synchronous communication. If it is known that one side (slave) is always ready for communication the other side may (master) communicate without waiting for synchronization. The *Shared* interface models the behavior of a volatile memory.
In order to simulate and verify the system-level design an executable description of the system is required. The industry standard for executable system level designs is SystemC, but the semantics of SystemC do not match the semantics of

our formal model perfectly. SystemC, as well as many other high-level modeling languages employed in the industry, are primarily software programming languages. For example, SystemC is used by a framework of class hierarchies and macro definitions to describe the structure of hardware systems, with the associated behavior being modeled in C++. While C++ has clearly defined semantics as a programming language, the high-level objects defined in the SystemC class framework lack a precise semantics with respect to the abstract hardware designs they are intended for. We solved this by restricting SystemC to a subset of certain constructs called SystemC-PPA.
The following example provides an idea on the semantic understanding of the system-level model for blocking message passing. Assume the component to design is a CPU, most designers will describe a CPU as a set of modules e.g., ALU, RegisterFile, Control Unit., which are connected to each other via ports. For example, the *Control Unit* has an output port next_instruction and the *ALU* has an input next_instruction. The *Control Unit* sends a new instruction formalized as a message (e.g. ADD rs1,rs2,rd) to the *ALU*. If the *ALU* is still busy with a different instruction the *Control Unit* is blocked until the *ALU* is ready for the next instruction. The blocking message passing handshake is also referenced as *rendez-vouz communication*. (Rendez-vouz communication is an analogy: In order for two people to communicate they have to be at the same place at the same time. If either one is missing the other one has to wait.)

3.1 Modules

Listing 2 shows the code of a SystemC module. It's composed of the constructor, ports, variables and the method containing the behavior named FSM. Those constructs are sufficient to describe any abstract hardware model. Every C++ construct which is not mentioned here, is not part of the subset and will be detected as unknown by the tool.
In the following sections each element is discussed in more detail.

```
1  struct Example: public sc_module{
2    //Constructor
3    Example(sc_module_name name):
4      var(9){SC_THREAD(fsm);}
5    SC_HAS_PROCESS(FPI_Master);
6
7    //Ports
8    blocking_in<int> b_in;
9    shared_out<bool> s_out;
10
11   //Variableless
12   int value;
13
14   //FSM
15   void fsm();
16 }
```

Listing 2: Example of a SystemC module

As shown in Listing 3, the user is also free to use the standard macros that are provided with SystemC. The compiler replaces those macros what results in the code shown above.

```
1 SC_MODULE(Example){
2   //Constructor
3   SC_CTOR(Example):
4     var(9){SC_THREAD(fsm);}
5   [...]
6 }
```

Listing 3: Example of a SystemC module

3.2 Variables and Datatypes

Variables are defined within the class definition. The initial value is set within the constructors initialization list. Every variable defined within the class will be added to the abstract model as a variable. Variables are only allowed to be declared within the class definition.
The only allowed built-in datatypes are *bool*, *int* and *unsigned int*. Line 12 in Listing 2 shows the declaration of an integer variable called *value*. The designer is also allowed to use enums or define it's own datatypes. The enum has to be defined within the scope of the module as shown in Listing 4 on line 2. Line 9 shows the declaration of an enum variable. Custom datatypes are declared as a struct with no constructor and no methods as shown in line 4. They are called *compound datatypes*, because they are composed out of a set of built-in datatypes and enums. For example, *Msg_type* defines a compound with three subvariables *addr*, *data* and *mode*. Until now, there is no possibility to assign

an initial value to the subvariables. Hence, they are initialized with the default value of the datatype. The custom datatypes are added to the abstract model as datatypes.

```
1  //Enum Datatype
2  enum Transfer_mode{read, write};
3  //Compound Datatype
4  struct Msg_type{int addr; int data; Transfer_mode mode;}
5  //Module
6  struct Example: public sc_module{
7     [...]
8     //Variables
9     Transfer_mode trans_mode;
10    Msg_type message;
11    //FSM
12    [...]
13 }
```

Listing 4: Varibles and custom types

3.3 Constructor

Constructors in C++ allow a large degree of freedom. For describing abstract hardware models, the constructor shown in Listing 5 is sufficient. Only the constructors initialization list is used for variable initialization. The constructor body is not taken into consideration.

```
1  struct Example: public sc_module{
2     //Constructor
3     Example(sc_module_name name):
4        port("port"),
5        var(9){SC_THREAD(fsm);}
6     SC_HAS_PROCESS(Example);
7     [...]
8  }
```

Listing 5: Example of a constructor

The SystemC macro SC_THREAD(fsm) registers the method fsm as a thread and the macro SC_HAS_PROCESS(Example) makes the module visible to the SystemC Scheduler. This functionality is purely SystemC specific and has no meaning for the abstract model. Each port of the module has to be initialized within the constructor as shown in line 4 for *port*.

3.4 Ports

Listing 6 shows all allowed port types. For example, line 4 shows the declaration of a blocking input port that receives a message of type integer. A port declaration always follows the structure *"interface_direction <message_type> name;"*. Possible interfaces are *blocking*, *shared*, *slave* and *master*, all of which implement a different blocking mechanism. Allowed directions are *in* for receiving and *out* for sending a message. As message type all previously defined custom types and built-in types are allowed. The interface defines how the communication is modeled on the RTL, hence they play a crucial role in the methodology. More detailed discussion follows in Section 3.5.

```
1  struct Example: public sc_module{
2     [...]
3     //Blocking interface
4     blocking_in<int> blocking_in;
5     blocking_out<int> blocking_out;
6     //Shared interface
7     shared_in<bool> shared_in;
8     shared_out<bool> shared_out;
9     //Slave
10    slave_in<Msg_type> slave_in;
11    slave_out<Msg_type> slave_out;
12    //Master
13    master_in<Transfer_mode> master_in;
14    master_out<Transfer_mode> master_out;
15    [...]
16 }
```

Listing 6: Example of all port interfaces

The ports are actually sc_ports with a custom interface. For example *slave_in* is defined as "using $slave\_in = sc\_port < slave\_in\_if < T >>$" and $slave\_in\_if$ defines the interface methods for this port. Declaring simple names for the interfaces facilitates code parsing and opens up the methodology to a wider audience as less SystemC knowledge is required.
Unfortunately, we couldn't use any standard SystemC ports, because as non of them provides pure RendezVouz communication. Initially the idea was to use an zero-depth sc_fifo for modeling RendezVouz communication, but the SystemC standard forbids zero-depth fifos. Furthermore, we restricted the users ability to use any other ports, because this would require an analysis of the underlying mechanisms. This has shown to be a complex tasks and doesn't support the PDD.

3.5 Interfaces

3.5.1 Blocking

The blocking interface implements the *rendezVouz* communication as mentioned earlier. In order to send a message, there are two interface methods *write*(*value*) and *nb_write*(*value*). The interface that implements this behavior is *rendezvous_out_if*<*T*>. By calling *port_name→nb_write(value)* the sender sends a message with the specified value. Prior sending it's checked whether the receiver is ready to receive a new message. This is achieved with the help of an internal flag of the interface that stores the status of the receiver. The sender is blocked until the reader_notify event is raised by the receiver.
By calling *port_name→nb_write(value)* the port also sends the message, but doesn't block the module. The sender assumes the receiver to be ready for communication. If the assumption is correct, the message is passed and *nb_write*(*value*) returns true. Otherwise, *nb_write*(*value*) returns false and the message is lost. Hence, by using this interface losing a message is accepted.

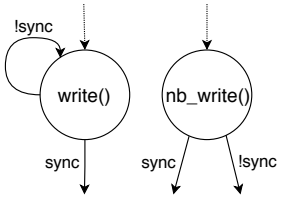


Figure 2: Transitions for write and nb_write

Figure 2 shows the different wait automatons for the regular write and the non-blocking write. The boolean synchronization signal *sync* is *true* if the counterpart is ready for communication, *false* otherwise. The figure shows the difference in the blocking behavior of those two modules. For receiving a message the methods *read*(*variable*) and *nb_read*(*variable*) are used. Upon handshake completion the message send by the sender is stored in *variable*. The receiving interface works similar to the write, with one difference for the *nb_read*(*variable*). If no new message is present the value of variable remains unchanged.

3.5.2 Shared

The shared interface offers a method *port_name→set(value)* for sending a message and a method *port_name→get(value)* for retrieving a message. This interface doesn't implement any handshaking mechanism and models the behavior of a volatile memory. The input and output of those ports may change at any timepoint. This interface is mainly intended for the case, that the system needs to talk with the environment. For example, if a value from a analog-digital converter is read, a handshaking mechanism is not necessary because the value changes continuously and the system reads whatever value is present at the current timepoint. The same ideas applies for sending value to a environment device. The shared interface behaves almost the same as a RT signal.
Because the interface doesn't implement a handshaking mechanism there is no wait state necessary for using a shared port. The shared ports become part of the datapath and are internally treated like variables that are visible from the outside/inside. Hence, the shared ports are going to show up in the datapath of the generated properties.

3.5.3 MasterSlave

This is a special interface that is only allowed to be used on synchronous chips. The common use-case for this interface is synchronous communication between two modules. The system-level designer knows that a message send/received by master can't be lost because the corresponding slave is always ready to communicate. This behavior is enforced by the later generated properties.

```
1  struct Example: public sc_module{
2     [...]
3     master_in<Transfer_mode> master_in;
4     master_out<Transfer_mode> master_out;
5     [...]
6  }
```

Listing 7: Example of all port interfaces

On system level this interface is modeled similar to the blocking interface, because we're only interested in passing the message. The master is allowed to communicate at any timepoint and it's ensured that the slave is always ready. No message from a master to a slave is lost and there is always a fresh message from a slave to a master. The slave accepts loosing a message, for example a *slave_out* port writes a message at each timepoint and accepts that the master doesn't receive it's message. A *slave_in* port always tries to receive a message.
There are some restrictions compared to the blocking interface. The interface is split into two sides, the master side and the slave side. Each master has to be connected to a slave on system level.
Once a module has a slave port, it's considered a slave-module and it's required that:

- every slave port is used within the fsm
- no slave port is used twice before every other slave-port is used
- the order in which the slave ports are used has to remain the same for each possible path
- ports with blocking interface are not allowed for slave modules

The master interface offers *port_name→write(value)* method for sending a message and *port_name→read(var)* for reading a message. There is no nb_read/nb_write necessary, because it's known that the counterpart is always ready to communicate and thus the return value evaluates to true.
The slave interface offers *void port_name→nb_write(value)* method for sending a message and *bool port_name→nb_read(value)* for reading a message. For the slave_out port the write method doesn't block,because loosing the message is accepted and the master is not required to catch every message. For the slave_in port a nb_read method is necessary because the message should only be captured if the corresponding master_out did actually send a message. The return value of nb_write is void where nb_read returns true if there is a new message from a master.

3.6 FSM

Basic
The supported set of SystemC language features allows ESL modeling of any type of digital hardware. The mentioned modeling restrictions may forbid the use of certain SystemC constructs, but they warrant the precise semantics of the models and they are key to enabling a sound, top-down refinement, "property-first" design methodology.
Listing 8 shows the basic structure of the method describing the behavior. When modeling the behavior, the user needs to adhere to the code structure shown in the example: There must be a single function that is registered as the one and only SystemC thread (cf. line 5) in this module.
The function must contain one outer infinite loop written as *while (true)* { ...} with the body describing the sequencing of sections of behavior, as shown in the example. This is necessary to precisely define a finite state control behavior in the form of an FSM. Because a module only has one possible behavior the user is only allowed to use one thread for describing the behavior. In order to simulate the blocking behavior using threads is mandatory, because sc_process and sc_method don't allow to be blocked.
Hardware does execute forever and thus the behavior is described within an infinite while(true) loop and at the end of the loop it's recommended to have a sc_zero_time. This is more a detail that is necessary due to the scheduler, because the other threats should be able to advance in their execution, too. The described model should describe untimed behavior and thus all other notions of time are not allowed.

```
1  struct Example: public sc_module{
2    [...]
3    //FSM
4    void fsm(){
5        while(true){
6        [...] //Functional description here
7        wait(sc_zero_time);
8        }
9    };
10 }
```

Listing 8: Behavior description

Listing 9 shows a simple behavioral description, that guides as a simple overview of allowed constructs within our SystemC-AML subset. In line 4 a message from the input port *blocking_in* is read by calling the interface method *read()*, the value of the message is stored in an integer variable *frames_ok*.
The execution of the thread is blocked until the counterpart sends a message, after the message is received execution continues at line 5. If *frames_ok* is larger than 10, a success counter is increased by one in line 6. At line 7 a message

is send over the blocking port *blocking_out* using the non-blocking interface. The port will offer a handshake to it's counterpart. If the counterpart is ready to communicate, the boolean variable success will evaluate to true and the message is passed, otherwise success evaluates to false.
The shared output *shared_out* sets it's value to true if the communication was successful. As mentioned in Section 3.5 for the *shared in* the port connected to *shared_out* reading from this port is same as accessing a variable inside this module and setting the value of this port is the same as setting the value of a variable. If frames_ok is less then 10 succ_cnt is reset to zero.

```
1  void fsm(){
2      while(true){
3      //Read blocking port
4      blocking_in->read(frames_ok);
5      if(framework > 10){
6        ++succ_cnt;
7        success = blocking_out->nb_write(succ_cnt);
8        shared_out->set(success);
9      }else succ_cnt=0;
10     wait(sc_zero_time);
11     }
12 };
```

Listing 9: Example1

A full simulation model of this example is provided in doc/Example1/ , this also applies for all other examples within this documentation. The module called Stimuli models the environment of the module Example1. The modules are connected by the code in sc_main.cpp. A more detailed view on the connection is provided in Section 3.7. In order to build the example follow the standard approach for building CMake projects.

Advanced

The only allowed control structure within the *while(true)*-loop are if-then-else branches. The use of for-loops and while-loops is not allowed, because everything within the behavioral description needs to be constantly evaluable. In order to generate the properties each path from a communication to a communication has to be of a finite length. Using unbounded loops is contrary to this requirement. It's possible to relax this requirement to bounded loops, but we provide a different mechanism for modeling loops, called **sections**.
Sections, are not only used for modeling loops but also for giving the generated properties a more meaningful name. For the example in Listing 9 all generated properties will have the same name appended by a unique identification number. We realized that most designers split the behavior in logical parts. For example, a module that waits for a start signal and then reads a specific amount of messages a designer would probably split the behavior in two parts *idle* and *reading*. Listing 10 shows the behavioral description of such a module. This example also shows how to use compound datatypes. The declaration of the type is found in Example2/types.h and an example for accessing the sub-variables is found in Example2/Stimuli.h.

```
1      //Sections
2      enum Sections{idle,reading};
3      Sections section,nextsection;
4
5      //Behavior
6      void fsm() {
7          while (true) {
8              section = nextsection;
9              if(section == idle){
10                 block_in->read(start_of_frame);
11                 if(start_of_frame) nextsection = reading;
12             }
13             else if(section == reading) {
14                 msg_port->read(msg);
15                 ++cnt;
16                 if(cnt > 4){
17                     nextsection = idle;
18                     cnt = 0;
19                 }
20             }
21             wait(SC_ZERO_TIME);
22         }
23     };
```

Listing 10: Example2

In order to model the idle phase and the reading phase, an enum "Sections" with two variables of this enum-type are declared in line 2 and 3 and initialized with *idle*. The execution starts at line 8 and continues with line 10. A message from port *block_in* is read and the message is a boolean value. In case a new frame is detected the section is changed to reading, if not the section idle is executed again. If the section changes to reading the next statement that is executed is line 14, which receives a new message. Afterwards a counter is increased by one and this is repeated until the counter

is greater than 4, in this case all the data is received and the module waits for the next *start_of_frame*. Hence, the module changed to section idle and resets the counter. The next execution of the loop will start with line 10 again.
A while loop is implemented as shown from line 9 to line 12 and a bounded for-loop from line 13 to line 20. The designer is free to use custom enums within the behavioral description to his convenience. If an enum with name Sections and variables section and nextsection are used, the tool will recognize this and store this information for later use. Furthermore, design with many branches are easier to process for the tool, if sections are used.
Best practice for writing modules is to have only one communication per section. This allows the tool to process large modules with many branches easily. Furthermore, the generated names of the properties are relying on section the communication the communication is described. It's also allowed to have sections without communication, in this case it's ensured by the tool that each possible path eventually ends at a communication. A path that doesn't end in a communication is reported by the tool with an error, because in this case no properties can be generated for this module. If no sections are used the tool adds a default section *run*.

3.7 Model

After describing the functionality of a single module we are now going to describe, how a system is built. In order to simulate a network of modules with SystemC each port of each module has to be connected to it's counterpart. The ports are connected through channels. In SystemC, channels are used to model communication and implement protocols. In our case the channels are implementing the communication.
In order to run the simulation, SystemC requires a function called sc_main that is the main function for the system designer. Within this main function the instances of the modules and the connection of the modules is taken care of. *At the current state, the tool doesn't support hierarchical networks. Hence, everything has to be declared on the top-level, which is the main.* We call the set of module instances and the connecting channels "Model".
Line 5 in Listing 11 shows the instantiation of the module "Example1" with the instance and variable name "example_module". The designer is free to create as many instances of modules as necessary and each module has it's own thread during the simulation.

```
1  int sc_main(int, char **) {
2      //Generating/Receiving messages
3      Stimuli stimuli("stimuli");
4      //Module
5      Example1 example_module("example_module");
6      //Channels
7      Blocking<int> blocking_channel2("blocking_channel2");
8      Shared<bool> shared_channel("blocking_channel");
9      Blocking<int> blocking_channel1("blocking_channel1");
10
11     //Connect example_module output to stimuli input
12     stimuli.block_in(blocking_channel1);
13     example_module.block_out(blocking_channel1);
14
15     //Connect example_module input to stimuli output
16     stimuli.block_out(blocking_channel2);
17     example_module.block_in(blocking_channel2);
18
19     //Connect shared ports
20     example_module.share_out(shared_channel);
21     stimuli.share_in(shared_channel);
22
23     sc_start(); //Start simulation
24     return 0;
25 }
```

Listing 11: Example1 sc_main

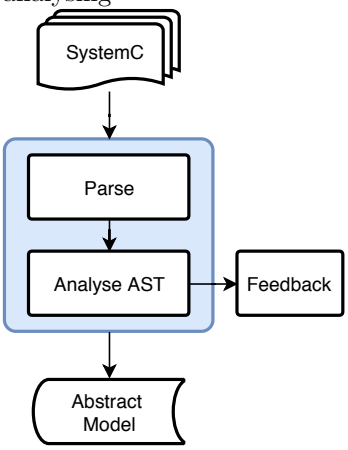
After creating the instances of the desired modules, the input and output ports have to be connected. The ports are connected through channels. We've created a channel for each interface and the user is required to use this channels in order to work with SCAM. Line 7 shows the declaration of a Blocking channel of datatype *int*. This channel is used to connect the input port "block_in" of module stimuli to the outport *block_out* of module example_module in line 12 and 13. For now it's required that each channel has exactly one input and one output.
After connecting each port to it's counterpart the simulation is started by using the macro sc_start(). The simulation runs until either no thread is executable anymore (deadlock) or the user stops the simulation, for example by calling sc_stop() within one of the threads. The sc_main.cpp may be used as input to SCAM.

4 Propderty-Driven Development

In order to work with the PDD in practice it's important that the system-level designer and the rt-level designer understand the required steps, as shown in Figure 1, in more detail. The first step is the refinement and analysis of the system-level model, afterwards the properties are generated and lastly, the hardware is designed. Step one and two are automatically executed by *SCAM*, this is why we are going to provide a basic idea of what happens within SCAM. The last step is explained by an example.

4.1 Step 1: Model analysis

Figure 3: Parsing and analysing

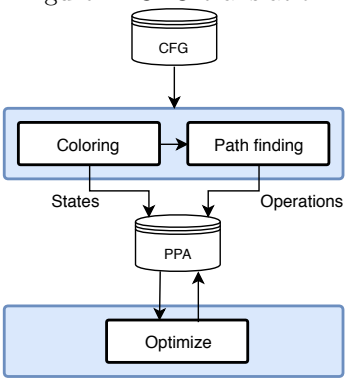


The refinement and analysis of the SystemC model is divided into two intermediate steps as shown Figure 3. The model is parsed and analyzed for compliance with the SystemC-PPA subset. Then structural and behavioral information is extracted and stored in an internal datastructure referred as abstract model (AM). If the designer uses statements that are not part of the subset warnings are generated and the model has to be refined accordingly.

4.1.1 Parsing and Analysing

In this step, the SystemC model is parsed by, the opensource compiler llvm/clang, as explained in [5]. The main reason for using clang is that the resulting abstract-syntax-tree (AST) implements a design pattern called visitor pattern. An AST stores program code in a tree-like structure and contains all static information related to the program including e.g., the SystemC-Scheduler. The combination of the AST and the visitor pattern enables an efficient analysis of the object structure.
Now, the AST is analyzed and all information that is required for property generation is extracted. In a nutshell: All C++ specific details, for example the SystemC scheduler, are stripped away and only information that is required to describe a module in terms of structure and behavior is stored in the AM. The structural information (e.g., ports and variables) are stored as an AST and the behavior (described by the thread) is stored in form of a control flow graph (CFG).
The tool analyses each object of the C++ program and checks for compliance with our SystemC-PPA subset. If an object violates the subset feedback, in form of warnings and errors, is generated and the objects are not stored in the AM. **Important:** The designer has to check the errors/warnings before implementing the RTL. It is possible, that the absence of objects in the AM alters the behavior described by the SystemC implementation.
For example, the line "*std::cout <<"Hello" <<std::endl*" produces the error "*-E- Unknown error: Stmts can't be processed(d)*". There are two possibilities for dealing with such an error. First, the designer is sure that std::cout pipes to the console. In this case the error is negligible. Second, the standard behavior of *std::cout* is changed and e.g., the stream pipes to a variable. The variable has an effect on behavior of the module. In this case the design behavior is different in absence of the statement. The designer has to refine the SystemC description to match the subset, otherwise the generated properties are meaningless.

Figure 4: CFG translation



The code for parsing and analyzing can be found *src/ParseSystemC* and the abstract model is described in *src/Model*. The class *src/ParseSystemC/ModelFactory* is the core element for creating the abstract model. The user is granted access to the AM by writing a backend. A detailed description on writing your own backend is found in *src/Backend/*.

4.1.2 CFG translation

A major benefit of the PDD is that a sound relationship between a system-level model and a RTL implementation is established. *TL: Elegantere Formulierung* Two models are considered sound if they show the same I/O behavior for the same input sequence. The CFG describes the behavior only in terms of the control flow and contains the I/O behavior implicitly. In order to generate the operation properties a statemachine that describes the I/O behavior explicitly is required. This is why the CFG is transformed into a statemachine, called PPA, with states being communication calls at the system level and transitions being a path in the CFG between two communication calls. Each transition describes one operational path of the CFG between to communication points. The transitions are referred as operations and each operation translates into a operation property.

Figure 4 shows an schematic overview of the transformation from a control flow graph to a PPA. The transformation is separated into two steps: *Coloring* involves finding the communication states and *Path finding* elaborates execution paths between two communication states. The PPA is then optimized regarding size and complexity, described by the step *Optimization*.

For explaining the details of the transformation the example of Figure 4.1.2 is used. The module describes a framer hardware that searches for the start of a data frame. This behavior is described in the section idle. As long as no new frame is detected the hardware stays in the section idle. After the detection of a new frame a shared output is set to true and the module changes to section start. The module stays in this state until the master out port did send the messages 3,2, and 1. Then the module switches to the section frame_data and reads 15 frames. After completion the module goes back to the section idle and waits for start of a new message.

Figure 5: SystemC example complex

```
1  enum status_t {in_frame, oof_frame};
2  struct msg_t {status_t status; int data; };
3  SC_MODULE(Example) {
4  SC_CTOR(Example):
5    nextsection(idle){SC_THREAD(fsm)};
6  enum Sections{idle,frame_start,frame_data};
7  Sections section,nextsection;
8  blocking_in<msg_t> b_in;
9  master_out<int>m_out;
10 shared_out<bool> s_out;
11 int cnt;bool ready;msg_t msg;
12 void fsm(){
13  while(true) {
14   section = nextsection;
15   if(section == idle) {
16    s_out->set(false);
17    b_in->read(msg);
18    if(msg.status == in_frame){
19     s_out->set(true);
20     nextsection=frame_start;
21     cnt = 3;
22     }
23   }else if(section==frame_start){
24    m_out->write(cnt);
25    cnt = cnt - 1;
26    if (cnt == 0) {
27     cnt = 15;
28     nextsection=frame_data;
29    }
30   }else if(section == frame_data){
31    ready = b_in->nb_read(msg);
32    if (!ready) {
33     m_out->write(msg.data);
34     if(cnt == 0){nextsection = idle;}
35     cnt = cnt - 1;
36    }
37   }
38  }};
```

The transformation requires several intermediate steps. The result of the first intermediate step is shown in Figure 9. Each node represents a line in the source code, e.g., *L.16*, stands for the line 16 in Figure 4.1.2. Line 16 is the first statement executed after reset. Execution continues with *L.17* and afterwards *L.18*. This statement is an if-then-else statement and if the condition evaluates to false, the section remains the same and execution continues with execution of *L.16*. This results in an edge from *L.18* to *L.16*. If the condition evaluates to true the execution continues with *L.19*.

The CFG of Figure 9 is obtained from the CFG of the thread *fsm()* provided by clang in Step 4.1. It's known that the thread is SystemC-PPA compliant and thereby follows a specific structure. Thus, it is possible to simplify the CFG by removing all CFG nodes for example the *while(true) loop*, the *if-then-else* for the sections and the *nextsection* assignment (if present). If such a node is removed from the graph, each incoming edge is merged with each outgoing edge. For example, the node for line 20 is removed from the CFG and *L.21* is redirected to the first statement of section *frame_start*. TL: Elegantere Formulierung Removing the nextsection assignment of line 34 is more difficult. The successor of *L.35* is dependent on the evaluation of *L.34*. If *L.34* evaluates to true execution continues with section *idle*, otherwise *frame_data*. In order to solve this problem, *L.35* is duplicated and one continues with section idle the other one with frame data.

4.1.3 Coloring and Path finding

The nodes that are marked red in Figure 9 indicate a communication call. In order to find the communication calls the graph is searched. Here, we are only interested in calls that implement a synchronization. The respective nodes are marked and transformed in a state for the PPA.

Now, all possible paths starting in a communication state and ending in a communication state are computed. Each paths describes an operation and covers an execution path between two communication calls at system-level. An operation results in a property and there is a property for each possible execution path at the system-level. These properties ensure also a correct transitioning between two I/O states at the register-transfer level. Thereby, it is ensured that the hardware shows the same I/O behavior as the system-level.

Figure 6: Operation P2

```
assume:
  state == L.17 and
  sync == true and
  msg.status==in_frame;
prove:
  state == L.24
  s_out == true;
  cnt == 3;
```

Properties are described as interval properties and are checked by the means of interval property checking (IPC) on the RTL design. A property has assumptions and expectations. If the assumptions are true (the operation *triggers*) the property checker proves

or disapproves the expectations.
For example, in state *L.17* there are two outgoing paths *P1*= $\{L.17 \rightarrow L.18 \rightarrow L.16 \rightarrow L.17\}$ and *P2*= $\{L.17 \rightarrow L.18 \rightarrow L.19 \rightarrow L.21 \rightarrow L.24\}$. *P1* is executed if the condition of the ITE of *L.18* evaluates to false, otherwise *P2* is executed.

Figure 7: Resulting PPA

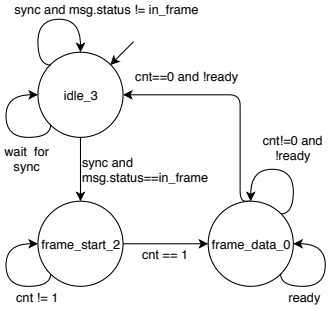


Figure 6 shows an abstract view of the operation *P2*. The assume part shows the trigger conditions, the component is in the control state *L17* and there is a new value (sync == true) and the message status is in_frame then the component has to: transition to state *L.24*, the value of the shared port has to be set to true and the counter is equal to 3.

4.1.4 PPA generation

The result of the previous is step is a set of states specifying communication calls at the system-level and operations describing transitions between states. The PPA describes a model in terms of its I/O behavior, but the part of the behavior that is implicitly defined by the handshaking of the interfaces is not part of the CFG and thus is not described by the set of properties, yet. For describing the communication the user is only allowed to use the provided interfaces. Thus, SCAM is completely aware of the underlying handshaking mechanisms. Each interface implements different handshaking mechanisms, which are added by SCAM to the set of properties automatically.
First, the CFG doesn't contain a reset operation. The **reset-operation**, is implicitly defined by the C++ semantics of the constructor. The ending state is the first communication after reset and the initial values result from the constructor and the path to the first communication. The path to the first communication has to be unique. Reading from a shared or master may result in multiple paths from the reset to the first synchronized communication call.
The **wait-operation** results from the communication of a blocking port. If *read() / write()* is called and the counterpart is not ready, then the module blocks. This behavior is implicitly defined through event based handshaking of the interface. The implementation is found in *src/Interfaces/*. SCAM adds a wait-operation to the PPA that enforces the module to remain in it's state until the counterpart is ready for communication.
The **non-blocking-operation** results from the communication of a blocking port and *nb_read() / nb_write()* is called. The success of the communication is returned by the communication call. Here the operation is split into two operations one for a successful communication and one for the unsuccessful communication.

4.1.5 PPA optimization

Initially, the PPA has four states, one for each communication call. Ports with the master interface form a special case, because they do not implement the two sided handshaking mechanism. A distinct state for a master is only required if the port is accessed multiple times in a row without passing a state with synchronization in between.
For example, if *L.26* evaluates to false the state *L.24* is accessed multiple times in a row. In order to send the messages (3,2 and 1) in the correct order the hardware remains in state *L.24* and changes the output accordingly.
For *L.33* this is not the case. The successor of *L.33* is either *L.31* or *L.17*. A distinct state is not required and the message of *L.33* is send concurrently with the synchronization of *L.31*. The state *L.33* is merged with the state *L.31*, this means that the operations L.31→L.33 and L.33→L.31 / L.33→L.17 are merged together to L.31→L.17 and L.31→L.31. Figure 8 shows the resulting operation for the merge of L.31→L.33 and L.33→L.31.
The resulting PPA is shown in Figure 7. In contrast to Figure 9, SCAM names the states not by the line of code they are declared in. Instead, states are named by section they are declared in extended by a unique id. If there are multiple communication calls within one section they are differentiated by the unique id. This is one of the main reasons why we emphasize the use of sections as much as possible. In absence of sections, a meaningful naming of the states is not possible and states maybe only identified by their unique number. In our example, *L.17* is renamed to *idle_3*, *L.24* is renamed to *frame_start_2* and *L.33* is renamed to *frame_data_0* and *frame_data_1* is missing because it is merged with *L.31*.

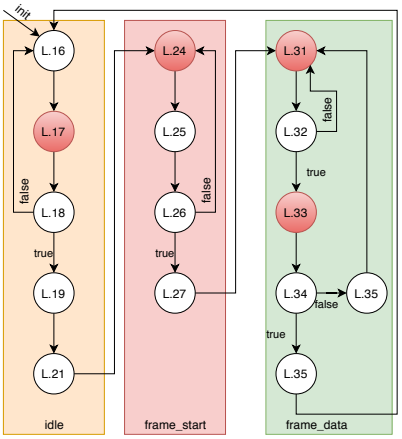
Figure 8: Merged operation

```
assume:
  state == L.31 and
  sync == false and
  ready == false and
  cnt != 0
prove:
  state == L.31
  cnt == cnt-1;
  m_out == msg.data;
```

Within Figure 7 there is an edge for each operation, including the reset. The edges show the trigger conditions, for simplicity the expectations are left out. The figure displays the difference between the interfaces. The state idle has a wait operation for the case of that the counterpart is not ready for communication where as the state frame_data_0 doesn't have a wait operation because the communication call uses the *nb_read()* method, which doesn't block. Frame_start doesn't have

any synchronization conditions, because the the interface is that of a master. The user has the option generate different views of the PPA in dot format by running SCAM with, e.g., -DOTfull or -DOTsimple.

Figure 9: Example CFG



The last optimization step is the reduction of the number of visible registers. At first, each variable at the system-level results in a visible register. If the variable is only used to store intermediate values, there is no need to latch the value and thus a register bears unnecessary overhead.
In our example this is the case for the variable *ready*, which stores the success of the nb_read() at line 31. The variable is assigned the value of the success of communication via port *b_in*. The trigger condition resulting from the if-then-else at line 32 is dependent on the evaluation of the communication. SCAM takes care of this by replacing each occurrence of *ready* with *b_in_sync* in each operation leaving *frame_data_0* Afterwards, it is checked whether the variable is used in any other operation. Because this is not the case, the variable is not part of the state variables and thus no visible registers is necessary.
For example, the variable *cnt* at line 35 is assigned it's previous value decreased by one. The new value is dependent on the previous and thus the variable becomes part of the state space of the PPA and a visible register is introduced.

Modules

Modules that are considered to be a slave module form a special case. The generated properties ensure that a message from a master is never lost. At the system-level, the interface is implemented using an one-sided handshake. The master port is blocked until the slave is ready.
In order to model this behavior at the RTL it is assumed that all slave ports communicate concurrently. Therefore, the tool merges sequences of slave communications into a single abstract state. A sequence here describes a path in the PPA where each of the slave ports is accessed exactly once. The access-order is of importance, it's ensured that each sequence accesses the ports in exactly the same order at the system-level. Here, the first sequence from reset is the reference for the other sequences.
It's important to remember that the slave out port will lose it's message if no master is waiting. Furthermore, the sync of the slave in port has always to be taken into consideration. It's possible that the slave may read before the master is ready!

4.2 Step 2: Property generation

The PPA is now stored as internal datastructure within SCAM. In order to generate the properties the user runs SCAM with the option -ITL or -SVA. This invokes a call to the respective backend found in *src/Backend*. Here, we are going to focus on the ITL property generation, the generation for the SVA properties follows exactly the same idea.
Before the properties are generated, the operation of the PPA have to be extended by timing and functions that enable connecting the properties to a RT implementation. In order to link the operations of the PPA to an RTL implementation we introduce *macros*. The designer is able to relate the abstract symbols of the system level to an arbitrary RT implementation by refining the macros.
The generated properties are formalized with the macros, which then get replaced by the parser with the according RT signals. For example there is a macro for the variable *cnt* and within the properties *cnt* is referenced by this macro. The designer has the task to refine the macro and thereby providing the information how *cnt* is implemented at the RTL. Macros are generated for the ports and the handshaking, the visible registers and the abstract states.

```
macro MACRO_NAME : RETURN_TYPE := MACRO_BODY end macro;
```

For filling the macros there are only three rules:

- Only finite sequences (of arbitrary length) may be evaluated
- Functions characterizing abstract inputs may be expressions over only input signals of the RTL design.
- Functions characterizing abstract outputs may be expressions over only output signals of the RTL design.

The name of the macro and the return type are provided by the tool. The designer has the task to refine the macro body according to the implementation. When SVA properties are generated the macros are exchanged by a function/define, but the idea remains the same.

4.2.1 Port Macros

In general for each blocking port macros for the notify, sync and datapath are generated. For the master interface a synch signal is not necessary and only the master_out port requires a notify signal and thus only the slave_in port requires a synch signal. The ports of type shared and slave_in require no synchronization. The datapath signal transports the actual message.

4.2.2 Visible Registers

The visible registers are part of the state space of the design. There is a visible register for each variable that is used at the system level, if the register is not removed in the previous optimization step. The macros for compound types are split into a separate macros for each subtype. For example, the variable *msg* is separated into two macros *msg_data* and *msg_status*.

4.2.3 Abstract States

The abstract states are derived from the communication calls at the system level. In our example we have three abstract states, idle_3, frame_start_2 and frame_data_0. The abstract states macros are of a boolean return type. If the hardware is in an abstract state the macro should evaluate to true, false otherwise.

4.2.4 Properties

We are going to split the generated properties into three different kinds of properties: reset operations, regular operations and wait operations.

Figure 10: Reset operation

```
property reset is
assume:
 reset_sequence;
prove:
 at t: idle_3;
 at t: cnt = 0;
 at t: msg_data = 0;
 at t: msg_status = in_frame;
 at t: s_out_sig = false;
 at t: b_in_notify = true;
 at t: m_out_notify = false;
end property;
```

The **reset operation** is a special operation, because it defines the state of the hardware after reset. It's really important to have a defined reset state, because otherwise properties may not cover all possible behavior.
Let's take a look at the reset property in Figure 10 from our example in Figure 4.1.2. The property is split into two parts: An assume part defining the trigger conditions and a prove part describing the expectations. One may read the properties as if-then relation. In this case: **if** *reset_sequence* **then** prove that *idle_3 and cnt=0 and ....* In state idle_3 the port *b_in* reads and thus the macro *b_in_notify* has to evaluate to true. At the same time, all other ports are not ready for communication and thus their notify signals have to evaluate to false.
The next property to look at is a regular **operation** as shown in Figure 11. This operation describes the path from line 17 to line 24 in the code. A value from port b_in is read and afterwards the port m_out writes the first value. This path is only taken if the condition at line 18 evaluates to true. The evaluation of this condition is depended on the value read from port b_in.

Figure 11: Regular operation

```
property idle_3_read_5 is
dependencies: no_reset;
for timepoints:
 t_end = t+32;
freeze:
b_in_sig_data_at_t=b_in_sig_data@t,
b_in_sig_status_at_t=b_in_sig_status@t;
assume:
 at t: idle_3;
 at t: (b_in_sig_status = in_frame);
 at t: b_in_sync;
prove:
 at t_end: frame_start_2;
 at t_end: cnt = 3;
 at t_end: m_out_sig = 3;
 at t_end: msg_data=b_in_sig_data_at_t;
 at t_end: msg_status=b_in_sig_status_at_t;
 at t_end: s_out_sig=true;
 during[t+1, t_end]: b_in_notify = false;
 during[t+1, t_end-1]: m_out_notify = false;
 at t_end: m_out_notify = true;
end property;
```

The property is segmented in different parts. It starts with the dependencies, here all we need is to assume that there is no reset, but the designer may add additional environment constrains to the properties. The next segment allows the designer to specify the length of an operation in number of clock cycles. The timepoint *t_end* specifies at which timepoint the prove part should be evaluated. Initially the value is set to *t+1*, in this case the property uses *t+32*. The reason for this is explained within the next section. Besides the constraints, the designer is only allowed to change the value of *t_end*. Everything else in the property has to remain untouched. Changing the properties may result in a loss of completeness !
Next, the values of the input signals are "freezed" at *t*, this means that the value of the signal is stored in a variable at timepoint *t*. In general, all values of the inputs and variables are captured at timepoint *t*, the values of the outputs are set at timepoint *t_end*.
The assume part specifies the assumptions (trigger conditions), which are all evaluated at timepoint *t*. The first condition is that hardware is in state *idle_3*, which means that the port

b_in waits for a new message. The operation is only triggered if a new message is available and the *b_in_sync* macro evaluates to true. The case that there is no new message is covered by the respective wait-operation. The last condition of the assume part is more interesting, because it differentiates from what the designer may expect by looking at the SystemC-PPA. The condition results from line 18, the difference is that the value is not checked against the variable *msg.status*. Instead the condition is based on the macro *b_in_sig_status*. At system-level the value of a variable changes instantly upon assignment. At the register-transfer level the value of a register changes with the next clock event. In order to evaluate the condition upon handshake completion the value of the input signal has to be evaluated. SCAM propagates the input signals through the paths.

The last segment is *prove*, which defines the next state of the hardware. All changes of the state variables and outputs are proven at time point *t_end*. In this case the hardware transitions from state *idle_3* to *frame_start_2* and the signal *cnt* takes the value 3. The variable *msg* stores the message, which is freezed at timepoint *t* in the freeze-variables *b_in_sig_data_at_t* and *b_in_sig_status_at_t*. In order to ensure that registers are updated to the value of *b_in* at *t* not at *t_end* As mentioned earlier, outputs are set at *t_end* where as inputs are read at *t*.

A **wait operation** is generated for each blocking message transfer. There are only two conditions for a wait operation the state and that the respective sync signal is low. The operation then proves that hardware remains in it's state. Thus the state variables, outputs and the abstract state have to remain the same. Furthermore, the property is only one cycle long. This is due to fact that the respective port already offers a handshake and an incoming sync should not be missed. This property stays entirely unchanged within the refinement process.

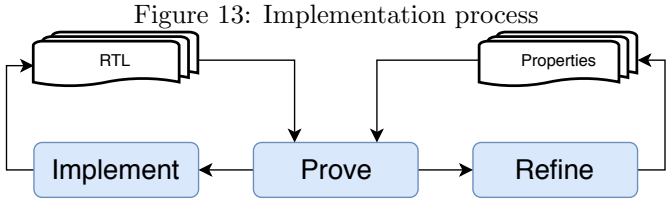
A special case are the **slave modules**. Here all generated properties are forced to be exactly one cycle long and thus (due to the state merging) every slave communicates once in each clock cycle and is always ready for it's master port.

Figure 12: Wait operation

```
property wait_idle_3 is
dependencies: no_reset;
freeze:
cnt_at_t = cnt@t,
msg_data_at_t = msg_data@t,
msg_status_at_t = msg_status@t,
s_out_sig_at_t = s_out_sig@t;
assume:
 at t: idle_3;
 at t: not(b_in_sync);
prove:
 at t+1: idle_3;
 at t+1: cnt = cnt_at_t;
 at t+1: msg_data = msg_data_at_t;
 at t+1: msg_status = msg_status_at_t;
 at t+1: s_out_sig = s_out_sig_at_t;
 at t+1: b_in_notify = true;
 at t+1: m_out_notify = false;
end property;
```

4.3 Step 3: Implementation and Refinement

Figure 13: Implementation process



Now we are going to explain the last step of the PDD that is the implementation of the hardware and the refinement of the properties. As an example we use Figure 4.1.2, which is considered to be the golden model for the hardware design process. Figure 13 provides an overview of the desired flow, which consist of three major steps: refining, implementing and proving. An iteration of the flow starts with choosing a property to implement, implementing the property and in parallel refining the properties accordingly. If the chosen property holds on the design the flow starts from the beginning.

The minimum requirements for the first iteration of the flow are a RTL template implementation, a set of properties and a property checker. The template has to implement the desired I/O interface, but is not required to have any behavioral elements. The macros for ports and visible registers of the set of properties have to be refined according to the RTL template. Finally, the template and the properties are loaded with a property checker. The first property that should be implemented is the *reset property* and the implementation and properties are refined such that the reset property holds.

Now, the iterative implementation process starts. First, a property to implement is chosen by the designer. The hardware designer has to understand the property and then decide on a hardware implementation for this property. In parallel the set of properties is refined according to the new implementation.

Lastly, the property checker checks whether the desired property holds on the design and thereby proves whether the hardware is correctly implemented for this property. If the property holds, then the designer should check whether all previously implemented properties still hold. If this the case then the next property is implemented. Otherwise, the designer uses the information from the debugger to either change the hardware implementation or refine the properties. This depends on whether the counter example is a *false negative* or a *true negative*.

In general it's recommended to start implementing all properties that start in the important state after reset. Then, all properties from the state after reset are implemented. The process continues until all properties hold on the design.

4.3.1 RTL template

In order to start the PDD flow the designer has to provide a RTL template implementation in form of an existing implementation, a custom RTL template or a template generated by SCAM. By running *SCAM* with parameter *-VHDL* a structural description of the module is created. There is a port for each port at the system level transporting the message, ports for synchronization (*sync* and *notify*) and signals for the visible registers. The template doesn't have any behavioral elements, although it provides a reset sequence that works with the reset from the properties. The designer is free to change anything in this template to his convenience.
The generated template also provides a package that provides a declaration of datatypes used on the system level. These types are also used in the properties and if the designer decides to have a manual type declaration it's his task to provide these to the property template. For example, an enum type declared at the system level is represented by the symbolic values of this enum type within the properties. This largely increases the readability of the properties. If the designer decides to implement the enum type manually there has to be a mapping to the symbolic type of the system level.
We explain, by the means of an example, how the abstract ports at the system level are mapped to an arbitrary RTL implementation. The generated template, from the golden reference, has a port declaration *b_in*, that transports a message of type *msg_t*. This port provides the message as a 32-bit integer for *msg.data* and a 1-bit boolean for *msg.status*.
Now, it is assumed that the entire message is not provided by a 33-bit input port. Instead, there is a 1-bit input receiving the message sequentially over multiple clock cycles. The RTL template is adjusted to this new environment by exchanging *b_in* with a port that provides only a 1-bit input *data_in*.
These are manual design decisions that are abstracted at system level. The generated properties are based on the abstract objects of the system level. The designer has to specify how these objects are implemented at the RTL by refining the properties accordingly.

4.3.2 Port and Visible Register refinement

After creating the RTL template, the properties have to be refined according to this template. Initially, the macro bodies of the generated properties are empty and the designer has to refine the macros according to the RT design. For refining the macros three cases are possible as shown in *Paper.vhi*:

1. The macro is not required, because a RT component with the same name exists and the macro is removed (e.g., line 1 - 4).
2. The macro is required, the RT signal has a different name (e.g., line 41)
3. The macro is required, the value of the macros evaluates over an arbitrary length or conditions (e.g., line 19 - 30).

No we are going to take a closer look at the macros for *b_in.data* and *b_in.status*. These two macros demonstrate the strength of PDD, because the cycle- and bit-abstract exchange of a message at the system level is transformed into a cycle- and bit-accurate implementation. The fact, that the message is transported sequentially by a 1-bit port is unknown at the system level. This is a manual RTL design decision and is a picture perfect example why automatic synthesis from the ESL to RTL is not feasible.
At first, we are going to take a look at the refinement of the macro *b_in_sig_data*. This macro describes the transmission of the data part of the abstract message. The protocol of the sequential interface is: *b_in* waits for a new input (outgoing *b_in_notify* evaluates to true) and the counterpart sends a new message (incoming *b_in_sync* evaluates to *true*). Hence, a new transmission starts and the first bit that is buffered is the bit at the time *b_in_sync* evaluates to true. The transmission finishes with the 32th bit.
The initial refinement of inputs for proving the reset property only requires to a syntactically correct refinement. The inputs don't influence the state after reset. The actual refinement is required when the first property after reset is implemented.
First, we are going to explain what happens at the RTL. Then property refinement is explained. The design is split into two processes. The process data_buffer (line 128 to line 140) continuously buffers the input stream of data_in and the process control implements the PPA.
The current state of the design process is that the reset holds and the first operation to implement is *idle_3_read_5*, which is implemented by line 56 to line 76. At first we are going to take a look at the trigger conditions. The hardware has to be in the important state idle_3 and the incoming *sync* is true. In idle_3 the counter *buffer_cnt* is initially 0 and

starts counting upon transmission start. The transmission is finished if the counter reaches the value 31 and thus 32 values of data_in are buffered.
The abstract view on this operation is that a transmission starts at timepoint *t*. The first value that is latched into buffer is the value of data_in at *t* and the last value is the value of data_in at *t+31*. This are in total 32 1-bit values the transmission is over and the prove part of the property is checked. The prove part of the properties specifies that after 32 clock cycles *msg_data* has the value of *b_in_sig_data* at *t*.
Now, we assume that the implementation step for this operation is done and the correctness of the implementation is to be checked. In order to do so, it's required to refine the behavior of the hardware within the properties. Let's take a look at line 9 to line 17 in *Paper.vhi*. Here the buffering behavior of the input port is described.
Eventually, the macro has to return the 32-bit input from the system level, but in reality the message is composed of 32 1-bit values provided by data_in. Thus, the macro is described by a concatenation of values from data_in at different timepoints. Because the value of *b_in_sig_data* is captured at *t* we need to look in the future in order to describe the upcoming values of *data_in*. ITL provides a *next* operation that enables to look an arbitrary amount of cycles in the future. The MSB of the integer is the value of data_in at *t* and the LSB is the value of data_in at *t+31*.
Now, we are going to take a look at the second part of the message, the status bit described by the macro *b_in_status* (line 19 to line 31). At the system level this information is transmitted as a 1-bit input upon handshake completion. At the RTL it's implemented over a sequence of time and thus the properties have to describe the RTL behavior in order to link the abstract system-level objects in the properties to the concrete implementation.
The value of the status bit is dependent on the last four input bits of *data_in*. If the sequence is equal to *'1111'*, then the status bit evaluates to *in_frame* and *oof_frame* otherwise. Within the macro body an ITL method *prev()* is used that allows to look into the past.
The latching of the values starts after the reset. If there is a reset within the last four cycles then the macro is only dependent on the values received after the reset. For example line 22 describes the case that there is a reset at *t-2*. The buffering starts after the reset and thus the only value of data_in that is latched is the value at *t-1*. Line 29 is the general case without a reset.
At this point, we advise the reader to change these macros and elaborate on the counter examples. Initially, debugging the counter example is confusing because it's unusual view on the hardware design process. The question the designer has to answer is: "*Are my properties wrong or my hardware?*". The easiest way, to elaborate on this is to follow what the hardware is doing for the provided trace. This answers the question on false or true negative quickly.

4.3.3 Abstract State refinement

In this section, basic idea on the refinement process for abstract states is provided. Here, the designer has to specify, which state bits of the global state vector describe the abstract state. In general, the designer is free to describe the abstract to his convenience, within the scope of our experiments two approaches showed the best results:

- *Output-based refinement:* The notify macros are used in order to describe the current state. Each abstract state implements a communication and thus notify signals are set/unset. This enables describing the abstract state with an one-hot encoding of the notify signals. If there are multiple system-level communication calls of same port the encoding is extended by additional conditions in order to distinguish between these calls.
- *State-based refinement:* The abstract state is described only dependent on internal state variables. This approach has wide range of possibilities.

In the following the state-based refinement is explained by an example for *idle_3*. The presented approach works really well if the designer sticks to the best practice recommendation of having a distinct section for each communication. As mentioned earlier, the implementation starts with an RTL template provided by SCAM. This template already provides a section signal which can be used to distinguish the abstract states.
Initially, the macro *idle_3* is refined with *section=idle*. But this leads to a counter example, where *buffer_cnt* is not zero upon transmission start. This is a false negative, because this state is unreachable from reset. The macro is extended to "*section=idle and buffer_cnt=0*" in order to start from a reachable state.

4.3.4 Timing

Initially, all operations are assumed to one cycle long and *t_end* is trivially defined *t+1*. In our example, the operations that implement the transmission of the message are actually 32 cycles long. The designer has to provide this information by changing *t_end* to *t+32*.

5 Installation

Download the most current version of SCAM:

```
git clone git@bordeaux.eit.uni-kl.de:SCAM
cd SCAM
git fetch --all
git pull origin master
```

Before installing SCAM, open install/install.sh and provide the path to SCAM, CMake and Python at the top of the file afterwards run the shell script. The binary will be copied to bin/ and if space is an issue the installation folder may be removed afterwards.
Requirement for installing SCAM are:

- CMake, minimum 3.0
- unzip
- g++, minimum 4.8

6 FAQ

- **The property generation takes a very long time/doesn't finish:** Please try to split up your design in various sections. This helps the tool during property generation. The best practice is to put each communication call into one section.
- else with only one undetected statement -¿ CFG error, usually people there is a case that should never occur. People throw an exception or write an assertion for this case. Obviously this doesn't translate to hardware.
- multiple path possible from init: usually after state merging the reset sequence is dependent on an input value
- Properties with compound/enum types: If your design uses compound/enum types they will also be used within the properties. The user has two options either define the types themselves or run SCAM with -VHDL and extract the package definition defining all user-defined types.

References

[1] J. Urdahl, D. Stoffel, and W. Kunz, "Architectural system modeling for correct-by-construction RTL design," in *Forum on Specification and Design Languages (FDL)*, (Barcelona, Spain), September 2015.

[2] J. Urdahl, D. Stoffel, and W. Kunz, "Path predicate abstraction for sound system-level models of RT-level circuit designs," *IEEE Transactions On Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 33, pp. 291–304, Feb. 2014.

[3] J. Urdahl, S. Udupi, T. Ludwig, D. Stoffel, and W. Kunz, "Properties first? a new design methodology for hardware, and its perspectives in safety analysis (invited paper)," in *The IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2016.

[4] "Scam."

[5] A. Kaushik and H. D. Patel, "Systemc-clang: An open-source framework for analyzing mixed-abstraction systemc models," in *Specification Design Languages (FDL), 2013 Forum on*, pp. 1–8, Sept 2013.