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

Manual — Version 0.1

Tobias Ludwig

2018-08-14

Contents

1 Introduction

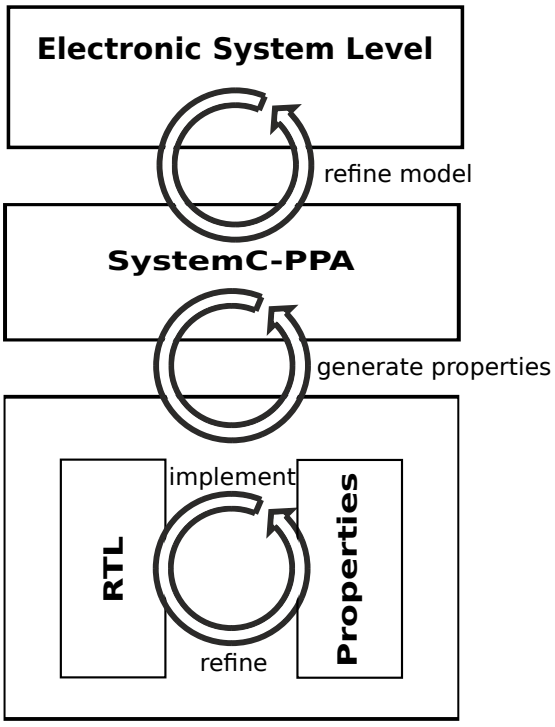
This manual describes *Property-Driven Design* (PDD), a new top-down hardware design methodology. It borrows ideas from so-called *Test-Driven Development* (TDD) [1], a software design paradigm aiming at writing high-quality code while maintaining high productivity. Before we delve into PDD, let us first review TDD and discuss some important aspects of it.

The core idea of TDD is that before you write code for a function you need to write the test for that function first. Writing tests guides the software engineer through the design process. At any point during the design process, the written code is covered by some test. Initially, the tests are very abstract, as are the implementations of the functions. The functions are refined concurrently with the tests, adding more and more behavior, until the final scope and functionality has been reached. The flow for designing a new feature with TDD is:

1. Create tests that describe the desired 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 when all tests hold for the design. This approach allows for an incremental software design process and results in fewer design bugs, complete test coverage and a documentation of the code.

Figure 1: Overview of the desired flow



Property-Driven Design (PDD) takes this idea to hardware design. Tests are replaced by formal properties. However, the designer doesn't need to write the tests. Instead, abstract properties are generated automatically from a high-level description. The PDD flow starts with a verified description of the component at the *Electronic System Level* (ESL). From this description the tests are generated automatically in form of interval [2] properties (e.g., SVA, ITL)by our tool *SCAM* ("**S**ystem**C** **A**bstract **M**odel"). When writing RTL code, the designer only needs to refine the properties by adding information about data types and clock cycle-accurate timing. The general flow is similar to TDD: Properties are refined step by step. The designer chooses a property and implements the hardware that fulfills the property. Once all properties hold on the design the hardware design process is finished.

The design entry point of PDD is the ESL. Figure 1 shows an overview of the desired flow which consists of three major steps. The first step is analyzing the 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*.

Once the properties are there, the RTL design phase begins. Property by property is refined and VHDL code is written that implements the behavior described by the property. Properties are checked, debugged and eventually proven using commercially available property checking technology.

One novelty of PDD is that the generated properties ensure that the RTL design has the same I/O behavior as the ESL design. As explained in our papers on PDD, the methodology ensures that the ESL description is a formally proven, sound abstraction of the RTL design. This allows the designer to use the ESL design as the golden model and verification results at the system level hold for the RTL without further proof.

This has the advantage that global design decisions can be made already at the electronic system level. If the ESL design works correctly so will the RTL design implemented from it. However, also any system-level bug will appear in the RTL. As ESL and RTL are no longer decoupled, it is important that the ESL is thoroughly verified before PDD-based RTL implementation is started.

Section 2 will provide a step-by-step-walk-through of PDD for a simple example, followed by a more detailed explanation of PDD in the subsequent sections. The *installation* of the software tool *SCAM* is explained in Section 5. All example files are available on our GitHub page [3].

2 Walk-Through

2.1 Step 1: ESL Analysis and Refactoring

```
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   //Variables
12   int value;
13
14   //Finite State Machine
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 PDD-methodology starts with a system-level design serving as golden reference for the following design process. Here, SystemC is used as system-level design entry language. A system is composed of a set of communicating modules. At the system level the main focus is on describing and verifying the *communication*

between modules. Each of these modules is later transformed into a set of properties.

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

The behavior of the module is described within the method *fsm* that is registered as a thread with 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*, otherwise *false*.

As you might already suspect from examining listing 1, it is not possible to use arbitrary SystemC constructs when following the PDD paradigm. Only a certain subset of SystemC is allowed, and the code needs to follow a certain structure and coding rules. The reason is that SystemC, per se, does not have a clear semantics with respect to cycle-accurate RTL implementation. The semantics need to be introduced by restricting the use of language constructs to what we call the "designable subset" of SystemC. This subset will be described and detailed in the following.

Let's continue with our first walk-through of PDD. We assume that our SystemC module has passed verification on the ESL and is considered ready for implementation. The SystemC code is analyzed using *SCAM*. We run the tool from the command line as `SCAM <path-to-file> -AML` The following example shows the output of SCAM when no errors are found:

```
############################
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 as EBNF. We use this DSL as an intermediate format in our flow.

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!
```

For example, using a `std::vector` on system level is beneficial when it comes to executing the simulation on the CPU but it is 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 must now make a decision,

- whether or not the statement is important for the module behavior,
- whether the SystemC description has to be modified,
- or whether the statement can be safely ignored as it has no effect on the behavior (e.g., a `std::cout`).

2.2 Step 2: Property Generation

Once the SystemC-PPA code is stable and deemed ready for RTL implementation, we can generate the initial set of abstract properties using *SCAM*. This is the second step of PDD in which we generate the properties with our tool *SCAM*. A single SystemC module or a system of SystemC modules serves as the input to the tool. The tool parses the files and generates an 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` flag invokes property generation in ITL (*InTerval Language*). ITL is a property specification language available for the OneSpin [4] property checker. We support it because the language is quite intuitive and apt to specifying the kind of properties used in PDD. Of course, also other property specification languages can be used in PDD. Currently, besides ITL, *SCAM* also support SystemVerilog Assertion (SVA) [5]. For SVA generation run the tool with the option `-SVA`. For a full list of possible commands run `./SCAM -help`.

In this section, we will provide only a quick overview over the generated abstract properties. For a more detailed explanation refer to Sec 4. In order to link the abstract objects of the system level to a concrete RTL implementation we introduce *macros*. The RT designer refines these macros by replacing 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 := MACRO_BODY end macro;
macro b_out_sig : bool := MACRO_BODY end macro;
```

There are two types of macros – synchronization macros for event signalling, and data path macros for transporting the message content. The synchronization macro names have the suffixes `sync` and `notify`. Thes signals are of Boolean type and implement the four-phase handshake. The generated data path macro

names have the suffix `sig`. The return type of these macros is the type of the transported message. The designer specifies, by refining these macros, how the abstract message is encoded at the RTL.

The protocol for sending (receiving) a message with a four-phase handshake is:

- The writer (reader) set the outgoing notify signal to *true*.
- The writer (reader) waits until the incoming sync signal evaluates to *true*.
- The message is exchanged by writing (reading) the datapath signal.
- After message exchange, the writer (reader) unsets the notify signal
- End of the transmission.

At the system level the handshaking is implicitly implemented through events and, thus, not visible to the user, whereas at the RTL there has to be an explicit implementation of the handshaking.

In our methodology, the generated properties capture the full I/O behavior of the system-level design. The system-level thread is transformed into a special finite state machine (*FSM*). This FSM represents a *path predicate abstraction* (*PPA*) [6, 2, 7] of the RTL design to be implemented. The states of the FSM represent the points of communication between modules, and the transitions between states represent the computations inside the module, as specified by the properties.

Listing 1 contains three communication calls (line 17, 19, 20). For each of these calls, a state is created. These states are called *important states*, because they represent important control points of the hardware.

When refining the property suite, the designer needs to fill in the corresponding state macros in order to specify what conditions in the RTL hardware represent the states. In order to do so it is 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;
```

There is a directed edge between two important states if there exists an execution path between the corresponding communication calls at the system level. Each edge translates into an interval property describing the actions taken in the RTL when moving from one important state to the next. The following property starts in the important state `run_0`, which is the state after reset.

```
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;
```

The property run_0_read_0 describes the execution path between line 17 and line 19 of Figure 1.

If the input is available ($b\_in\_sync == true$) and the input value is greater or equal 11 the hardware moves 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 property assumes that:

- The hardware is in state `run_0`,
- a new input is available ($b\_in\_sync == true$),
- and the value of the received message is greater or equal 11 (`b_in_sig` ≥ 11).

Then, it is proven that:

- The hardware moves to important state `run_1`,
- the value of `b_out_sig` is set to *true*,
- and the outgoing message is offered by raising `b_out_notify`.

The hardware remains in state `run_1` until the recipient of the message accepts the handshake and the message is passed.

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

Now, the RT designer has the task to specify how the abstract system-level objects are implemented by the RT-design by filling in the macros. For example, the macro `b_in_sig` represents the incoming value. The designer has full freedom in how to implement a representation of this data. The designer maps the corresponding RTL signals to the property by providing a body for this macro.

Let us suppose that the incoming message 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 could also be that a message is transported by two different RT signals and the resulting value is the sum of those two signals. The designer would, accordingly, provide this information in the macro body:

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

In case there exists 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 data types and a bare-bone VHDL implementation matching the properties. The template is not a synthesized design and thus doesn't contain any behavioral components.

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 refines the important state macros for the currently implemented operations during the design process.

3 System Level and SystemC-PPA

The system-level model is executable. It is composed of modules that communicate with each other. Communication is modeled on the transaction level, i.e., the system behavior is that of time-abstract and world-level descriptions of finite state machines sending each other messages based on synchronization events. Each module is modeled as a PPA describing the behavior as an FSM in terms of its I/O states and transitions. The behavior at the system level results from an *asynchronous product* of the individual FSMs. This allows for a modeling of all interleavings of messages being passed between the modules, and ensures capturing the full behavior of the system-level model. Due to the untimed behavior of the system-level model, each module is allowed to run at its own speed. In order to exchange a message between two modules they need to synchronize through a handshake.

At system level this handshake is implemented through events. During the RTL design process the handshake is realized by a four-phase handshake.

Sometimes, implementing full four-phase handshaking bears unnecessary overhead, e.g., in cases where loosing a message is acceptable. SystemC-PPA, therefore, provides three different kinds of communication interfaces called *ports*. The three supported interfaces, in our experience, provide enough flexibility to handle any hardware communication needs. The type of communication interface is selected during the system-level design process.

Each interface generates a different kind of property suite and, therefore, affects the hardware design process. The basic interface is called *blocking* and implements a 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 (the *slave*) is always ready for communication then the other side (*master*) may communicate without waiting for synchronization. Finally, 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 needed. 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 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 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. Assuming 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 sometimes also referred to as *Rendezvous communication*. (Rendezvous 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 is composed of the constructor, ports, variables and the method containing the behavior named *fsm()*. These constructs are sufficient to describe any abstract hardware model. Every C++ construct not mentioned in this document 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   // Variables
12   int value;
13
14   // FSM
15   void fsm();
16 }
```

Listing 2: Example of a SystemC module

As part of the SystemC library macros for generating modules and constructors are provided. The designer is free to implement the modules with or without using these macros. Listing 3 uses the provided macros `SC_CTOR` and `SC_MODULE`, where as Listing 2 uses standard C++ class and and constructor definitions.

```
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 Data Types

If a module requires variables for its behavioral description, they have to be declared as part of the SystemC module. The declaration of local variables within the behavioral description is not allowed. The initial value is set within the constructor's initialization list and the allowed datatypes for variables are the built-in types: *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 custom datatypes. The enum has to be defined within the scope of the module, as shown in listing 4 on line 2. Line 9 demonstrates the declaration of an enum variable. Custom data types are declared as a struct with no constructor and no methods, as shown in line 4. They are called *compound data types*; their members consist of built-in data types and enums. For example, *Msg_type* defines a compound with three sub-variables *addr*, *data* and *mode*. Every variable defined within the class will be added to the abstract model. The custom data types are added to the abstract model as data types. The template generator prints them as a record type in VHDL.

In the current version of *SCAM* it is not possible to assign an initial value to the sub-variables of a compound type. This will be fixed in a future version.

```
1  //Enum Data Type
2  enum Transfer_mode{read, write};
3  //Compound Data Type
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: Variables and custom types

3.3 Constructor

For describing abstract hardware models, the constructor shown in Listing 5 is sufficient. The constructor's initialization list is used for variable initialization. The only allowed parameter is the module name. The constructor's body is not taken into consideration. More parameters are not needed because the module has to be statically analyzable. Not taking the macro body into consideration is a restriction in the current version of *SCAM* that will be fixed in the future.

```
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. The macro `SC_HAS_PROCESS(Example)` makes the module visible to the SystemC Scheduler. This functionality is purely specific to SystemC 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.

3.4 Ports

Listing 6 demonstrates all of the 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. Message types may be any of the built-in types as well as enums and compound data types. 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 following information is for advanced SystemC users. This section may be skipped, if the reader desires to use SystemC-PPA, without the need to understand the implementation details. A *port* is, actually, an `sc_port` 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.

3.5 Interfaces

3.5.1 Blocking

The blocking interface implements the Rendezvous communication as mentioned earlier. In order to send a message, there are two interface methods `write(value)` and `nb_write(value)`.

By calling `port_name->write(value)` the sender sends a message with the specified value and is blocked until the reader is ready.

This interface is used for an asynchronous system in order to ensure synchronization between the modules before message transmission. A blocking interface guarantees that a message is never lost.

In addition to the regular blocking communication mechanism, this interface also allows for non-blocking sending and receiving of messages. A non-blocking send is initiated by calling `port_name->nb_write(value)`. The port sends the message whether the receiver is ready or not, but doesn't block the module. If the message is successfully delivered, the function call returns *true*. If it is lost, the return value is *false*.

Please note that even though this interface is called "blocking", it contains functions for both, blocking *and* non-blocking read and write.

Figure 2: Transitions for write and nb_write

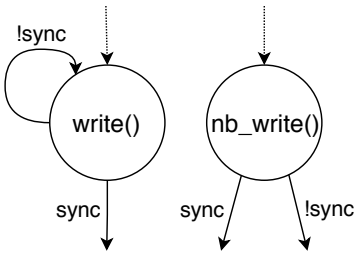


Figure 2 shows the different wait automatons for the regular write and the non-blocking write. The Boolean synchronization signal *sync* is *true* if the counterpart is ready for communication, *false* otherwise. The figure shows the difference in the blocking behavior of those two modules.

For receiving a message, the methods `read(variable)` and `nb_read(variable)` are used. Upon handshake completion the message sent 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 received the value of variable remains unchanged.

3.5.2 Shared

The shared interface offers two methods: `port_name->set(value)` for sending a message, and `port_name->get(value)` for retrieving a message. This interface doesn't implement any handshaking mechanism and therefore, is able to model the behavior of a volatile memory. The input and output of these ports may change at any time point. This interface is needed mainly to model unordered input data like sensor values.

For example, if a value from an 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 time point. The same idea applies for sending a value to a peripheral device.

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 in *synchronous* hardware systems, i.e., systems using a common clock. The interface is split into two sides, the *master* side and the *slave* side. The common use case for this interface is synchronous communication between two modules for which the system-level designer knows that a message sent/received by a master cannot be lost because the corresponding slave is always ready to communicate. This behavior is enforced by the properties generated later for this construct.

```
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

Each master has to be connected to a slave. The master interface offers the `port_name->write(value)` method for sending and `port_name->read(var)` for reading a message. The slave interface offers *void* `port_name->nb_write(value)` for sending and *bool* `port_name->nb_read(value)` for reading a message. For the slave_out port the write method doesn't block, because possibly 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*. Method nb_read returns *true* if there is a new message from a master, and *false* otherwise.

The semantics of this interface is that the master is allowed to communicate at any time point with the guarantee that the slave is always ready. No message from a master to a slave is lost and there is always a new message from a slave to a master. The slave accepts loosing a message, for example a *slave_out* port writes a message at each time point and accepts that the master doesn't receive its message. A `slave_in` port always tries to receive a message. At the system level the master is blocked until the corresponding slave is ready for communication whereas the slave modules never block. Once a module has a slave port, it is considered a slave module, requiring that

- every slave port is used within the FSM,
- no slave port is used a second time before every other slave-port has been used,
- the order in which the slave ports are used has to remain the same for each possible execution path,
- no ports with blocking interface may be used if a module contains the slave side of a MasterSlave interface.

3.6 FSM

3.6.1 Basic Features and Example

The supported set of 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.

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 behavior, as shown in the example. This is necessary to precisely define a finite state control behavior in the form of a FSM. Because a module only has one possible behavior the user is only allowed to use one thread for describing the behavior. The use of threads is necessary for simulation of blocking behavior (because `sc_process` and `sc_method` cannot be blocked).

The continuous operation of digital hardware is modeled by the infinite *while (true)* loop. The model describes untimed behavior and thus all notions of time, except `sc_zero_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 providing an overview of allowed constructs within SystemC-PPA. 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*.

```
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

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 incremented by 1 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 its 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 its value to *true* if the communication was successful. If *frames_ok* is less then *10 succ_cnt* is reset to zero. As with all examples in this documentation, a complete simulation model is provided in directory `doc/Example1/`.

The module called *Stimuli* models the simulation environment of the module *Example1* and is not meant to be RTL-implemented using PDD. The modules are connected in `sc_main.cpp`. A more detailed view on the connection is provided in Sec. 3.7. In order to build the example, follow the standard approach for building CMake projects.

3.6.2 Advanced Features and Example

The only allowed control structure within the *while (true)*-loop are if-then-else constructs. The use of *for* and *while* loops is not allowed, because in order to generate interval properties covering a finite number of clock cycles each path from a communication function call to the next communication function call has to consist of a finite number of C statements. Unbounded loops violate this requirement. However, we can still model bounded loops using a a SystemC-PPA construct called *sections*.

This construct also helps structuring the behavior into individual operations. The properties generated by *SCAM* will refer to the section names to ease implementation and debugging.

For the example, in listing 9, all generated properties will have the same name appended by a unique

identification number which may make it difficult to distinguish the individual properties. However, if the designer places each communication function call in a separate section, the generated properties reflect the names of the section containing the calls. This allows to easily identify the described the execution path within the SystemC-PPA description.

Human designers usually decompose behavior into parts that reflect subfunctions or operations. Consider, 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 into two parts: *idle* and *reading*. Listing 10 shows the behavioral description of such a module. This example also shows how to use compound data types. 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" and two variables of this enum type are declared in lines 2 and 3 and are initialized with *idle*. The execution starts in line 8 and continues with line 10. A message of Boolean type is read from port *block_in*. In case a new frame is detected the *section* is changed to reading, if not the *section idle* is executed again. If control is transferred to *section* 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 has been received and the module waits for the next *start_of_frame*. Hence, the module changes to *section idle* and *resets* the counter. The next execution of the loop will start with line 10 again. Effectively, lines 9 to 12 implement a *while* loop, and lines 13 to 20 implement a bounded *for* loop. 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, designs with many if-then-else constructs 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 more easily. Furthermore, the name of a generated property is derived from the *section* in which the communication is described. It is also allowed to have *sections* without any communication.

SCAM is able to consider execution paths spanning more than one section, and guarantees that every possible path starts and ends at a communication function call.

If no *sections* are used at all the tool implicitly assumes a single default *section* called *run*.

3.7 Model

Now that we know how to describe the functionality of a single module we will look at how to put modules together to a system.

In order to simulate a network of modules with SystemC, each port of every module has to be connected with its counterpart. The ports are connected through *channels*. In SystemC, channels are used to model communication and to implement protocols.

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 modules are instantiated and connected with each other. The current version of *SCAM* does not support hierarchical networks. Therefore, everything needs to be declared on the top level, which is `sc_main`.

We call the set of module instances and the connecting channels "Model".

```
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

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. Each module has its own thread during the simulation. Module instances are interconnected through channels. There is a channel for each interface. *SCAM* requires that only these channels be used for communication between modules.

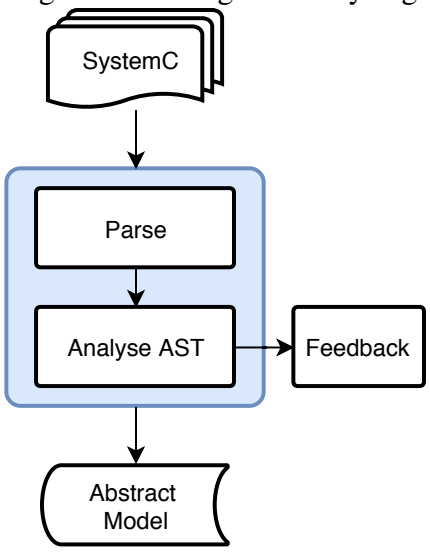
Line 7 shows the declaration of a blocking channel of data type *int*. This channel is used to connect the input port *block_in* of module `stimuli` to the outport *block_out* of module `example_module` in lines 12 and 13. With the current version of *SCAM*, it is required that each channel have exactly one input and one output. The simulation of the system is started by using the macro `sc_start()`. The simulation runs until there are no more executable threads(deadlock), or simulation is terminated, for example by calling `sc_stop()` within one of the threads.

4 Property-Driven Development

In order to work with the PDD methodology in practice, it is 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, after which the properties are generated and, lastly, the hardware is designed. Step one and two are automatically executed by *SCAM*. Therefore, in the following, we provide a basic idea of what happens insides SCAM. The last step is explained by an example.

4.1 Step 1: Model analysis

Figure 3: Parsing and analysing



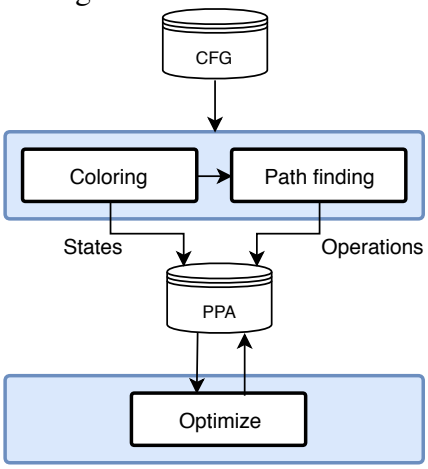
The refinement and analysis of the SystemC model is divided into two intermediate steps as shown in Figure 3. The model is parsed and analyzed for compliance with the SystemC-PPA subset. Then, structural and behavioral information is extracted and stored in an internal data structure referred as abstract 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 open source compiler llvm/clang, as explained in [8]. 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, for example the SystemC scheduler. The combination of the AST and the visitor pattern enables an efficient analysis of the object structure.

Next, 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).

Figure 4: CFG translation



The tool analyzes each object of the C++ program and checks for compliance with SystemC-PPA. If an object violates this SystemC 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 design.

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, which has an effect on the 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.

The code for parsing and analyzing can be found in *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 methodolgy is that a sound relationship between a system-level model and a RTL implementation is established. From a mathematical point of view, the relationship is sound with respect to LTL properties. This implies that any LTL property proven on the abstract model will also hold on the RTL model. From a practical point of view, any design property shown to hold on the system level will also hold on the RTL. Hence, the task of system verification can be moved from the RTL to the system level and the result will translate automatically to the RTL.

After parsing and analyzing the system-level model the behavior is stored as a CFG. In order to ensure soundness an explicit representation of the I/O behavior is required. However, the provided CFG describes the behavior in terms of the control flow and contains the I/O behavior only implicitly. The CFG therefore has to be transformed to a different graph, which describes the I/O behavior as a PPA. The PPA has an important state for each call to a communication interface at the system level and a state transition for each possible execution path between two calls (referred to as *operation*).

Figure 4 shows a schematic overview of the transformation flow from a CFG to a PPA. The transformation is separated into two steps: *Coloring* involves finding the important states and *path finding* elaborates execution paths between two important states on the CFG. The PPA is then optimized regarding size and complexity, described by the *optimization* step.

For explaining the details of the transformation the example of Figure 5 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 the start of a new

message.

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 5, which 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 is known that the thread is SystemC-PPA compliant and thereby follows a specific structure. Thus, it is possible to simplify the CFG by removing the *while(true) loop*, the *if-then-else* for the sections and the *next-section* 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*.

Removing the *nextsection* assignment of line 34 is more difficult. After *L.34* is evaluated *L.35* is executed. 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*. This problem is solved, as shown in Figure 9, by duplicating node *L.35* and connecting each instance to a distinct section *start*.

```
 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:         }}};
```

Figure 5: SystemC-PPA example

4.1.3 Coloring and Path finding

The nodes that are marked red in Figure 9 indicate a communication call. Initially the coloring is unknown and 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 to a state for the PPA.

Now, all possible paths between two communication states are computed. Each path results in a property, which checks the correct transitioning between two

I/O states at the RTL. In this way, it is ensured that the hardware shows the same I/O behavior as the system-level.

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 disproves 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 if-then-else of *L.18* evaluates to *false*, otherwise *P2* is executed.

Figure 6 shows an abstract view of the operation *P2*. The assume part shows the trigger conditions. The assumptions are that the component is in the control state *L17*, there is a new value (*sync* == *true*) and the message status is *in_frame*. Then, the expectations regarding the component are:

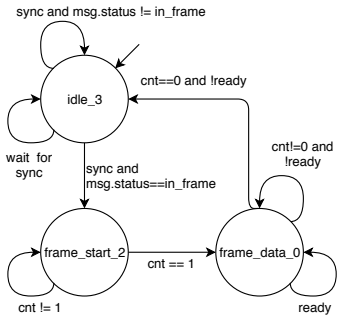
Figure 6: Operation P2

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

- transition to state *L.24*
- the value of the shared port evaluates to *true*
- the counter is equal to 3

4.1.4 PPA generation

Figure 7: Resulting PPA



The result of the previous step is a set of states representing 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 present in the CFG and, thus, is not described by the set of properties yet. To account for this, the user is only allowed to use the provided interfaces for describing communication, as SCAM is completely aware of their 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 port or master port 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 its 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.

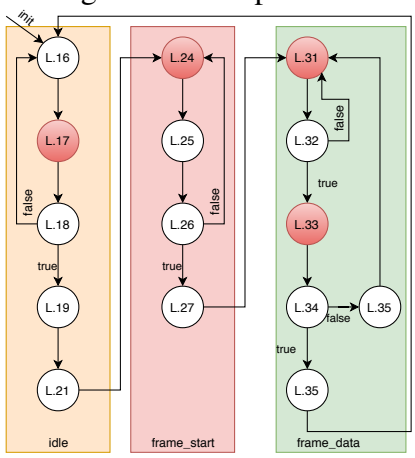
Figure 8: Merged operation

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

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 and 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 to use sections as much as possible. In absence of sections, a meaningful naming of the states is not possible and states may only be 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*. The state *frame_data_1* is missing because it is merged with *L.31*.

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 in case that the counterpart is not ready for communication whereas the state *frame_data_0* doesn't have a wait operation because the communication call uses the non-blocking *nb_read()* method. *Frame_start* doesn't have any synchronization conditions, because the interface is that of a master. The user has the option to 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 the given example, this is the case for the variable *ready*, which stores the success of *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. Since 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 its 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 a 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 important 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 has to be 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 is important to remember that the slave out port will lose its message if no master is waiting. Furthermore, the sync of the slave in port has always to be taken into consideration. It is possible that the slave may read before the master is ready!

4.2 Step 2: Property generation

The PPA is now stored as internal data structure 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*. In this section 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 operations of the PPA have to be extended by timing and functions that enable connecting the properties to an RTL 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 RTL implementation by refining the macros.

The generated properties are formalized with the macros, which then get replaced by the parser with the according RTL 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 as well as the important 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 sync signal is not necessary and only the *master_out* port requires a notify signal and thus only the *slave_in* port requires a sync 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 separate macros for each subtype. For example, the variable *msg* is separated into two macros *msg_data* and *msg_status*.

4.2.3 Important States

The important states are derived from the communication calls at the system level. In our example we have three important states, *idle_3*, *frame_start_2* and *frame_data_0*. The important states macros are of a boolean return type. If the hardware is in an important 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.

The **reset operation** is a special operation, as it defines the state of the hardware after reset. This is particularly important, 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 5. The property is split into two parts: An assume part defining the trigger conditions and an expect 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*.

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 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 dependend on the value read from port *b_in*.

The property is segmented into different parts. It starts with the dependencies, 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 time point *t_end* specifies when the prove part should be evaluated. The default value is set to *t+1*. In this case the property uses *t+32*. 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 in any other way may result in a loss of completeness!

Next, the values of the input signals are frozen at *t*, this means that the value of the signal is stored in a variable at time point *t*. In general, all values of the inputs and variables are captured at time point *t*, while the values of the outputs are set at time point *t_end*.

The assume part specifies the assumptions (trigger conditions), which are all evaluated at time point *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 condition is not described using the variable *msg.status*, instead the evaluation of the condition is based on the macro *b_in_sig_status*. At the system level the value of a variable changes instantly upon assignment. At the RTL 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.

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 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 frozen at time point *t* in the freeze-variables *b_in_sig_data_at_t* and *b_in_sig_status_at_t*. The registers are updated to the value of *b_in* at *t* not at *t_end*. As mentioned earlier, outputs are set at *t_end* whereas inputs are read at *t*.

Figure 12: Wait operation

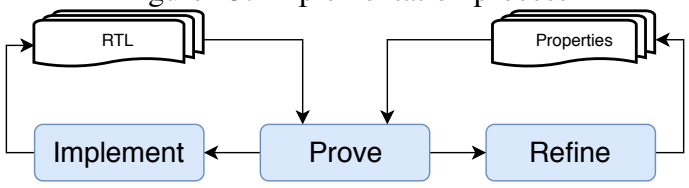
```
property wait_idle_3 is
dependencies: no_reset;
freeze:
cnt_at_t = cnt@t,
msg_data_at_t = msg_data@t,
msg_status_at_t = msg_status@t,
s_out_sig_at_t = s_out_sig@t;
assume:
 at t: idle_3;
 at t: not(b_in_sync);
prove:
 at t+1: idle_3;
 at t+1: cnt = cnt_at_t;
 at t+1: msg_data = msg_data_at_t;
 at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

A **wait operation** is generated for each blocking message transfer. There are only two assumptions for a wait operation: The hardware in a specific state and the respective sync signal is low. Then it is proven, that that hardware remains in the same state. Thus the state variables, outputs and the important states 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 its master port.

4.3 Step 3: Implementation and Refinement

Figure 13: Implementation process



The last step of the PDD methodolgy is the implementation of the hardware and the refinement of the properties. As an example, we use the design of Figure 5, 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: refinement, implementation and proof. 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 an 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.

As the last step in every iteration, the property checker examines whether the considered property holds on the design. It thereby proves whether the hardware is correctly implemented for this property. If the property holds, 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 to refine the properties. This depends on whether the counter example is a *false negative* or a *true negative*.

Generally it is recommendable to begin with 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 an RTL template implementation, either in form of an already 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/her convenience.

The generated template also provides a package containing the declarations of data types used on the system level. These types are also used in the properties. Hence, if the designer decides to have a manual type declaration it is his/her responsibility 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, via 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*.

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 the 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 RTL design. For refining the macros three cases are possible, as shown in *Paper.vhi*:

1. The macro is not required, because an RTL component with the same name exists and the macro is removed (e.g., line 1 - 4).
2. The macro is required, the RTL 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).

As an example, the macros *b_in.data* and *b_in.status*, describing the incoming message, are refined. The macro *b_in_sig_data* describes the transmission of the data part of the abstract message. These two macros demonstrate the strength of the PDD methodolgy, 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 always feasible.

The following section describes, the underlying hardware implementing the transmission and afterwards the refinement of the macro is explained. The behavioral part of the RTL design (*Paper.vhdl*) is split into two processes. The process *control* (line 40 to line 128) implements the automaton described by the PPA and the process *data_buffer* (line 128 to line 140) implements the buffering of the input stream of *data_in*. The protocol for a message transmission over the sequential port *data_in* is: The hardware is in the important state *idle_3* and the abstract port *b_in* waits for a new message (outgoing *b_in_notify* evaluates to *true*). A new transmission is started, if the corresponding counterpart sends a new message (incoming *b_in_sync* evaluates to *true*). Now, for 32 clock cycles, the 1-bit values of *data_in* are buffered. The first bit is buffered along with the start of the transmission, indicated by *b_in_sync* evaluating to *true*.

Note that the designer is dealing with incomplete implementations during the top-down design process. The assumed current state of the design process is: The reset property holds and the first operation to implement is *idle_3_read_5* (here implemented by line 56 to line 76). In order to implement the operation, the designer starts with 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 designer has to implement the expectations of the property, e.g., the visible register *msg_data* has to store the received message.

Here, an abstract view on this operation is provided, which is always implicitly given by a property. The transmission starts at time point *t* and the first value latched into the buffer is the value of *data_in* at *t* and the last value is the value of *data_in* at *t+31*. This sums up to total 32 1-bit values at which part the transmission is over and the expectation part of the property is checked. The expectations 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 is 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 integer 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 time points. Since the value of *b_in_sig_data* is captured at *t*, we need to look into 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 into 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*.

Next, we 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 is 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 protocol for the status bit is: The value 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 of *data_in* 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 is an 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 Important State refinement

In this section, a basic idea on the refinement process for important states is provided which specifies which state bits of the global state vector describe the important state. In general, the designer is free to describe the important states to his/her 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 important state implements a communication and thus notify signals are set/unset. This enables describing the important state with a one-hot encoding of the notify signals. If there are multiple system-level communication calls to the same port the encoding is extended by additional conditions in order to distinguish between these calls.
- *State-based refinement:* The important state is described only dependent on internal state variables.

In the following, the state-based refinement is explained by an example for *idle_3*. The presented approach works 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 provides a section signal which can be used to distinguish the important 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* counterexample, 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 be one cycle long and *t_end* is defined with *t+1* per default. 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
```

Make sure your environments provides the following tools:

- CMake, version 3.0 or higher,
- unzip,
- g++, version 4.8 or higher.

Before installing *SCAM*, open and configure the shell script `install/install_new.sh` by providing the paths to *SCAM*, cmake3 and python3 at the top of the file. Afterwards, run the shell script to compile and install *SCAM*. The binary will be located in the `bin/` folder.

References

[1] K. Beck, *Test Driven Development: By Example*. Addison-Wesley, 2002.

[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] "SCAM." https://github.com/ludwig247/SCAM.

[4] Onespin Solutions GmbH, "OneSpin 360 DV-Verify." https://www.onespin.com/products/360-dv-verify/.

[5] SystemVerilog Language Working Group, "IEEE standard for system verilog – unified hardware design, specification, and verification language," *IEEE Std. 1800-2009*, 2009.

[6] 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.

[7] 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.

[8] 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.