A Framework for Static Analysis of VHDL Code

Marc Schlickling
Saarland University & AbsInt GmbH
schlickling@cs.uni-sb.de

Markus Pister
Saarland University & AbsInt GmbH
pister@cs.uni-sb.de

Abstract

Software in real time systems underlies strict timing constraints. These are among others hard deadlines regarding the worst-case execution time (WCET) of the application. Thus, the computation of a safe and precise WCET is a key issue[1] *for validating the behavior of safety-critical systems, e.g. the flight control system in avionics or the airbag control software in the automotive industry.*

Saarland University and AbsInt Angewandte Informatik GmbH have developed a successful approach for computing the WCET of a task. The resulting tool, called aiT, is based on the abstract interpretation [3, 4] of timing models of the processor and its periphery. Such timing models are hand-crafted and therefore error-prone. Additionally the modeling requires a hard engineering effort, so that the development process is very time consuming.

Because modern processors are synthesized from a formal hardware specification, e.g., in VHDL *or* VERILOG*, the hand-crafted timing model can be developed by manually analyzing the processor specification.*

Due to the complexity of this step, there is a need for support tools that ease the creation of analyzes on such specifications. This paper introduces the primer work on a framework for static analyzes on VHDL.

1 Introduction

During the last years, embedded systems have become nearly omnipresent in everyday life. Embedded processors are used in a variety of application fields: health-care technology, multimedia applications, telecommunication, automotive and avionics, weapon guidance, etc. Common characteristics of many applications are that high computation performance has to be obtained at low cost and low power consumption. Moreover many applications have safety-critical characteristics and must satisfy hard real-time constraints. This leads to an additional requirement to be respected in embedded system design: the requirement of predictable performance. It is not enough for microprocessors to yield high peak performance, but it should also be possible to statically guarantee their worst-case performance. Contemporary superscalar architectures are characterized by deep complex pipelines, often with features like out-of-order execution, branch prediction, and speculative execution which make determining the guaranteed performance of applications a difficult task [8].

The worst-case execution time analyzer *aiT* originally developed by *Saarland University* and *AbsInt Angewandte Informatik GmbH* is a tool for computing safe and precise upper bounds of the worst-case execution time (WCET) of tasks. The computation is based on the *abstract interpretation* [3, 4] of *timing models* of the processor core and its system controller [18, 19]. The tool takes the executable as input and performs several static analyzes on it. The input is transformed into an intermediate representation called *Control-Flow Representation Language* (CRL)[2] [13], on which the analyzes are based. Further details about the *aiT* tool-chain can be found in [6].

The computation of the WCET of a task mainly depends on the so called *pipeline analysis* in which the behavior of the processor pipeline and the underlying system controller are modeled. This is done by abstracting from everything that is not needed for the timing behavior of the processor pipeline. Further details about how to create a pipeline analysis can be found in [18].

As of today, these models are hand-crafted and only obtainable with a hard engineering effort. Therefore, the development of a pipeline analysis is a very time consuming and error prone process. And the complexity dramatically increases along with each new processor generation used within embedded systems[3].

A formal processor specification (usually coded in a *Hardware description language* like VHDL or VERILOG) can be very helpful in the creation of a pipeline analysis. But even then, one needs to manually analyze the specification in order to find suitable abstractions. In order to ease this we introduce a framework for static analyzes of VHDL de-

[1] besides the functional correctness of the system

[2] In our framework we use the second version of this intermediate representation, called CRL2.

[3] As of today, these embedded processors are rather similar to and featureful as modern desktop processors.

scriptions. To this end, we developed a VHDL frontend that transforms the specification into the intermediate language CRL2 mentioned above. Different static analyzers can be generated using the *Program Analyzer Generator* PAG based on a formal analysis specification.

The paper is structured as follows: Section 2 contains a description of PAG and CRL2. Section 3 then details the semantics of VHDL. In Section 4, our analysis framework is described and illustrated with an example in Section 5. Section 6 shows some experimental results. Section 7 gives an outlook on future work and Section 8 concludes.

2 Preliminaries

We use PAG to generate static program analyzers based on a control flow graph. The next two sections introduce the basics of the control flow representation language and the Program Analyzer Generator.

2.1 CRL

The Control-Flow Representation Language (CRL2) was developed to provide an intermediate format that simplifies analyzes and optimizations on a control flow graph [13].

A *control flow graph* is a representation, using graph notation, of all paths that might be traversed through a program during its execution. Nodes in the graph represent basic blocks, i.e. straight-line pieces of code. Directed edges are used to represent jumps in the control flow.

Assuming that the control flow graph is always well-formed, the structure of CRL2 is hierarchically organized in instructions, basic blocks and routines. Thereby, the former is always completely enclosed within the latter. A sample control flow graph is given in Figure 1 showing two routines (`simul` and `environment`). The edge between the two routines indicates a call dependency, each routine call is represented through *call/return blocks* in the callee's routine. The call/return blocks ease interprocedural control flow analyzes [17] and are more or less placeholders for the branch to the called routine.

CRL2 is a very flexible language by virtue of an attribute-value concept, i.e. each element can be extended by attributes coding arbitrary information. To process a program represented by its control flow graph, PAG can be used.

2.2 Program Analyzer Generator

PAG is a powerful tool for generating program analyzers. Based on a high-level specification of a data flow problem, PAG automatically generates a program analyzer[4] which can be used in arbitrary applications [14].

The triple $(K, L, [\![]\!])$ is called a *data flow problem* for a complete lattice[5] L and a control flow graph K, if $[\![]\!] : N \rightarrow (L \rightarrow L)$ is a function assigning functions from $L \rightarrow L$ to the nodes of K. These functions are called *transition functions* and are used for updating the data flow value during analysis.

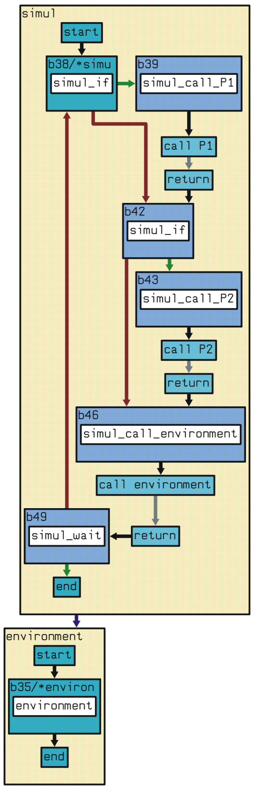


Figure 1: Sample CRL2-graph

More details on data flow problems can be found in [16], a description of the specification language for PAG in [1].

3 VHDL Semantics

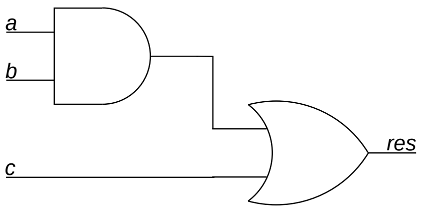
VHDL is an IEEE Standard defined in IEEE 1076 [2, 12]. The focus of the language ranges from specifying circuits at wavefront level to describing large system behaviors with high-level constructs. As a result, the standard is huge. The focus of this paper only considers the *synthesizable subset* of VHDL, defined in [11].

A VHDL description of a circuit consists of an interface declaration defining the in- and output signals of the circuit and of one or more implementation(s). In VHDL, the first is called an *entity*, the second an *architecture*. Figure 2 shows a simple 3-bit counter.

The implementation is given in form of two *processes* (`P1` and `P2`). Each process executes its code, whenever one of the *signals* contained in the processes *sensitivity lists* (*clk* and *rst* for `P1`, *cnt* for `P2`) changes its value. After execution of all statements, execution suspends until another

[4] in ANSI-C

[5] A *complete lattice* is a partially ordered set in which all subsets have both a supremum and an infimum. An example for a complete lattice is the power set of a given set, ordered by inclusion. The supremum is given by the union and the infimum by the intersection of subsets.




```
entity comb_logic is
  port(a,b,c:in std_logic;res:out std_logic);
end;
architecture rtl of comb_logic is
  signal wire: std_logic;
  component and_gate is
    port(u,v:in std_logic;w:out std_logic);
  end component;
  component or_gate is
    port(x,y:in std_logic;z:out std_logic);
  end component;
begin
  and_gate port map(a,b,wire);
  or_gate port map(wire,c,res);
end;
```

Figure 3: Composition of VHDL components

```
entity counter is
  port(clk:in std_logic; rst:in std_logic;
       val:out std_logic_vector(2 downto 0));
end;
architecture rtl of counter is
  signal cnt:std_logic_vector(2 downto 0);
begin
  P1: process(clk,rst) is
    if (rst='1') then
      cnt<="000";
    elsif (rising_edge(clk)) then
      cnt<=cnt+'1';
    end if;
  end;
  P2: process(cnt) is
    val<=cnt;
  end;
end;
```

Figure 2: 3-bit counter in VHDL

change of at least one signals value. Thus, the sensitivity list of a process is an implicit wait-statement at its end.[6]

VHDL also supports component-based circuit specifications. Figure 3 gives an example for hierarchical circuit composition. Here, the combinatorial function $res = a \wedge b \vee c$ is modeled using a logical-and and a logical-or gate. Having a hierarchical composed specification of a circuit, *elaboration* has to be performed in order to get a flat definition of it. Elaboration does all the required renaming for unifying names, wires all structural descriptions, etc. The result is one large entity consisting of a number of processes and some locally defined signals.

A VHDL process consists of a set of local *variables* that are only accessible from inside the process. By contrast, local signals can be accessed by more than one process, but only one process is allowed to drive the value of a signal.[7] Within a process, execution of statements is done sequentially.

VHDL makes a distinction between the assignments to a variable and to a signal. Assigning a value to a variable takes effect immediately (i.e., the next reference of this variable returns the newly assigned value), whereas the assignment of a value to a signal is only *scheduled* to be the future value (i.e., the next reference returns the old value). E.g., in Figure 2, the signal assignment `cnt<=cnt+'1';` schedules the next value of `cnt` to be `cnt` plus one, but the next reference `val<=cnt;` schedules the next value of `val` to be the *current* value of `cnt`. These future values take effect as soon as all processes *suspend* their execution. The semantics of a VHDL program, i.e. a set of processes, can be described as follows:

1. Execute processes until they suspend.
2. If all processes are suspended, make all scheduled signal assignments visible at once.
3. If there is a process being sensitive on a signal having changed its value, resume this process and go to step 1.
4. Otherwise, an external signal must change its value (e.g., the clock signal). If this happens, resume all processes waiting for this signal and go to step 1.

Thus, the semantics of VHDL can be seen as a two-level semantics: sequential process execution at its first, signal update and process revocation at its second level.

4 A VHDL analysis framework

As mentioned in the introduction, we present here a framework for statically analyzing VHDL code using *abstract interpretation*. The structure and data flow of our framework is illustrated in Figure 4. As input, we have the VHDL model that we want to analyze and a PAG specification for a static analysis. We developed a tool

[6] In VHDL, the use of explicit wait-statements and sensitivity lists is exclusive. We assume, that the only place within a process, where wait-statements may occur, is at the end of the body of a process.

[7] In full VHDL, *resolution functions* can be used for value computation of signals being driven by two or more processes.

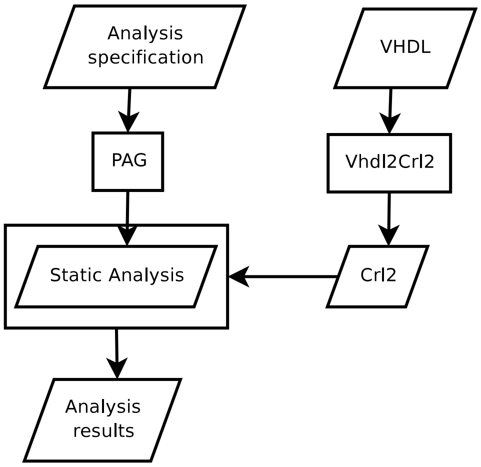


Figure 4: Structure of VHDL analysis framework

VHDL element	CRL2 element
Process, Function, Procedure, Concurrent signal assignment, Concurrent procedure call, Loop	Routine
Function calls Procedure calls	Routine calls Routine calls
Sequential statement	Instruction

Table 1: Mapping VHDL to CRL2

called `Vhdl2Crl2` that transforms the VHDL into semantically equivalent CRL2 constructs. From the analysis specification, PAG generates a static analyzer that works on the CRL2 description. And at the end, the analysis emits its results. The mentioned components in our framework are now detailed in the following subsections.

4.1 Mapping VHDL to CRL2

In order to express a VHDL description in semantically equivalent CRL2 constructs, we need to give a mapping from VHDL components to CRL2 components. Table 1 shows this mapping. Processes are transformed into routines as well as functions, procedures, concurrent signal assignments, concurrent procedure calls and loops. The transformation of loops to routines roughly means, that the whole loop body is moved into a newly created routine and the original location of the loop is replaced by a call to the new loop routine. This improves the quality of static analyzes of loops. More details on this so called *Loop transformation* can be found in [15].

The correspondence of function/procedure calls to routine calls as well as the mapping of sequential statements to instructions is rather intuitive.

4.2 VHDL as a sequential program

Regarding the special semantics of VHDL (cf. Section 3), we need to express the input VHDL description as a sequential program whose control flow is represented by CRL2. The reason for this is that we want to generate the analyzer itself from a concise PAG specification, where PAG is an analyzer generator for programs (cf. Section 2).

As mentioned in Section 3, variables in VHDL are process-local and processes run in parallel. Additionally signal assignments only take effect after all processes have finished their execution. These semantics directly induce that there are no side effects between the different VHDL processes. So, we can serialize their execution without changing the semantics of the whole model.

In order to formulate a VHDL description as a sequential program, we just need to choose an arbitrary execution order among the processes and iteratively execute them in this order. The program then consists of a routine, let's call it `simul` (cf. Figure 1), whose body contains routine calls, where each called routine represents one of the former VHDL processes. Additionally each such routine call for an original process it guarded with a conditional statement that evaluates the sensitivity list of the process. If at least one of the signals in the sensitivity list has changed, the call is taken. The result of such a transformation from a VHDL model into a sequential program is shown in Figure 1. Here, you can see the `simul` routine, containing routine calls for each VHDL process, namely for `P1` and `P2`. The basic blocks *b38* and *b42* contain an instruction `simul_if` that represents the guards for the sensitivity list evaluation. The basic block *b49* containing the instruction `simul_wait` is the so called synchronization point. Here, the signal assignments take effect and we can decide whether a signal value has changed compared with the previous iteration.

In Figure 1 there is another call instruction not mentioned so far, the `simul_call_environment` that calls the routine `environment`. We need this routine for analyzing open systems, which roughly are systems that have external input signals. These signals have to be set somewhere in the environment of the system modeled by the VHDL description and they drive the behavior of the whole system. One intuitive example for such an external signal is the `reset` signal.

4.3 Modeling the clock

If we want to analyze synchronous designs, i.e. systems that are synchronized with either the rising or the falling edge of the clock signal, we need to model this signal somehow. Unfortunately the synthesizable subset of the VHDL standard [11] does not enable us to model a clock signal[8].

[8] In synthesizable VHDL, there are two main restrictions: a process can not be sensitive on a signal it drives and there is no possibility to wait for a timeout. Thus, there is no construct left for modeling the frequent change of a clock signal.

Despite this, we can simulate the clock easily by introducing a new routine, called `clock`, that calls the routine `simul` twice. Before one call the clock signal is set to one and before the other call the clock signal is set to zero. This makes our approach rather flexible as we can analyze synchronous designs as well as asynchronous ones.

5 Example: Constant Propagation

This sections describes, how the framework introduced in Section 4 can be used to model a constant propagation analysis on VHDL.

A *constant propagation analysis* determines for each program point, i.e. each statement, if a signal or variable has a constant value, when execution reaches that point. To model this, we introduce a mapping from identifier names to their corresponding value and extend it by the usual bottom and top elements to denote not yet considered program points and unknown values respectively.

$$F \equiv (identifier \rightarrow (value \cup \top)) \cup \bot$$

As stated in Section 3, VHDL differs between signal and variable assignments. Thus, the domain of the data flow problem for constant propagation analysis has to cover *current* and *future* values of the identifiers used. Furthermore, to evaluate the condition of process guards (`simul_if`, see Section 4.2), it is necessary to decide, whether a signal has changed its value or not. Therefore, the domain has to be extended by the *old* values of signals. Thus, the domain *dfi* for the constant propagation analysis is:

$$dfi \equiv F \times F \times F$$

The transition functions for updating an incoming data flow value $dfi_{pre} = (cur_{pre}, fut_{pre}, old_{pre})$ to the output value dfi_{post} for the different nodes can be directly defined as follows:

- assignment node
  The data flow value for an assignment can be computed from the incoming value in case of a variable assignment by updating the current and the future value of dfi_{pre} with the newly assigned value or $\top$, if this value is statically not computable. In case of a signal assignment, only the future value has to be updated.
- `simul_if` node
  The guard encapsulates the sensitivity list of a process and is responsible for the repeated execution of it. A process is only re-executed, if one of the signals in its sensitivity list changes its value. This can be checked by comparing cur_{pre} with old_{pre}. Based on this result, the data flow value is propagated into the process or not.
- `sync` node
  At this node, all scheduled signal assignments take effect and are made visible at once. The new data flow value is computed by copying the future values to the current one and the current ones to the old ones.

  $$dfi_{post} = (fut_{pre}, fut_{pre}, cur_{pre})$$
- environment node
  Writing a special rule for this node allows us to analyze the VHDL code with respect to special system criteria. E.g., if we want to analyze the reset behavior of the code displayed in Figure 2, we introduce a rule $cur_{post} = cur_{pre} \backslash [rst \rightarrow 0]$ and $fut_{post} = fut_{pre} \backslash [rst \rightarrow 1]$ always setting the current and future value of `rst` to 0 and 1 respectively. This yields to the perception, that `cnt` and therewith `val` have constant values during reset.

Using PAG and the rules above yields in a constant propagation analysis on VHDL. The results can now be used for further analyzes and transformations of the VHDL code as described in Section 7.

6 Experiments

Despite the example given in the previous section, we successfully[9] tested our implementation with the complete VHDL specification of the Leon2 processor core [7]. The Leon2 is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture with a 5-stage pipelined integer unit, data and instruction cache, hardware multiply, divide and MAC units. The model is highly configurable, and particularly suitable for system-on-a-chip (SOC) designs. The VHDL specification of the Leon2 consists of more than 80 modules containing about 75.000 lines of code.

7 Outlook

The analysis framework described in this paper is only one part of a much bigger task: The semi-automatically derivation of timing analyzers from a formal processor specification.

With the work here, we are able to read VHDL specifications and to perform static analyzes like constant propagation on it. This eases the task of finding suitable abstractions for a processor model, i.e. cropping the model to the parts that are only relevant for the timing behavior.

As noted earlier, a VHDL design is too large to be used directly in timing analysis, thus we have to throw away things not influencing the timing. This can be achieved by slicing backwards from the place, where instructions leave the pipeline, i.e. finish execution. Only external signals and variables or those being assigned to in the slice can influence timing. Signals and variables used (i.e. read) but not assigned in the slice do not change their values (i.e. have fixed values) and can be omitted.

[9] Successfully here means, that we are able to create static analyzers based on analysis specifications.

Even after this removal, the model may be too large for timing analysis. Thus, we approximate concrete components by abstract ones (see [6] for more details) which are smaller in size. An example can be found in [5].
For this, we want to develop support tools to incorporate transformations on the VHDL model based on the analysis results semi-automatically in order to derive abstract models.
From an abstract processor model, we need to generate a C-code analysis fitting into the tool chain of the *aiT* tool.
At the end of the story, we want to derive a timing analyzer from a formal hardware specification. This not only speeds up the task a creating such a timing analysis but additionally the generated analyzer is already validated due to the derivation from the hardware specification.

8 Conclusions

Safety-critical applications as for example the flight control software in avionics or the airbag control software in the automotive industry are underlying hard real-time constraints. Therefore the computation of the worst-case execution time (WCET) is the key issue for guaranteeing that a system satisfies its timing boundaries.
The growing complexity of modern processor architectures used within such safety-critical systems complicates the task of creating a sound timing analysis. Currently, these analyzes are based on hand-crafted abstract processor models. But this is a very time consuming and error-prone process.
To ease the task of finding suitable abstractions, we introduced a framework for static analyzes of formal processor descriptions in VHDL. By transforming the VHDL model into a sequential program, we can generate static analyzers from a concise specification using the *Program Analyzer Generator* [14]. Our framework is very flexible because we can analyze open designs as well as closed ones, i.e. systems that does or does not depend on external driven signals respectively.
By using CRL2 as the intermediate representation, we can combine several analyzes. This means that an analysis can use the result of another one because the results are annotated as attributes in our intermediate representation.
To illustrate the practicability of our framework, we showed how to create a constant propagation analysis on a VHDL description of a 3-bit counter.
In [9, 10] Charles Hymans gives a design for static analysis of VHDL that uses abstract interpretation. Despite this and to the best of our knowledge, our work presented here is the first work concerning a framework for generating static analyzers on VHDL code. Our results are currently going to be used for the semi-automatically derivation of timing analyzers from a formal processor description in VHDL.

Acknowledgments

This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS) and by EU network of excellence IST-004527 ARTIST2 on Embedded Systems Design.

References

[1] AbsInt Angewandte Informatik GmbH. *The Program Analyzer Generator User's Manual*, 2002.
[2] P. J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.
[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
[4] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2-3):103–179, 1992.
[5] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, 1997.
[6] C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann. New Developments in WCET Analysis. In T. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation. Theory and Practice.*, volume 4444 of *LNCS*, pages 12–52. Springer, 2007.
[7] J. Gaisler. *Leon2 Processor User's Manual - Version 1.0.30*, July 2005.
[8] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proceedings of the IEEE*, 91(7), July 2003.
[9] C. Hymans. Checking safety properties of behavioral vhdl descriptions by abstract interpretation. In *SAS*, pages 444–460, London, UK, 2002. Springer.
[10] C. Hymans. Design and implementation of an abstract interpreter for vhdl. In D. Geist and E. Tronci, editors, *CHARME*, volume 2860 of *LNCS*. Springer, 2003.
[11] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard P1076.6 1999 VHDL Register Transfer Level Synthesis*, 1999.
[12] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard P1076 2000 VHDL Language Reference Manual*, 2000.
[13] M. Langenbach. CRL – A Uniform Representation for Control Flow. Technical report, 1998.
[14] F. Martin. *Generating Program Analyzers*. PhD thesis, Saarland University, 1999.
[15] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of Loops. In K. Koskimies, editor, *CC*, volume 1383 of *LNCS*. Springer, 1998.
[16] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
[17] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*. Prentice-Hall, 1981.
[18] S. Thesing. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.
[19] S. Thesing. Modeling a System Controller for Timing Analysis. In *EMSOFT*, pages 292–300, 2006.