A Framework for Static Analysis of VHDL Code

Marc Schlickling   Markus Pister

Compiler Design Lab
Computer Science Dept.
Saarland University

AbsInt GmbH
Science Park 1
Saarbrücken

7th Int'l Workshop on Worst-Case Execution Time Analysis

Outline

- Computation of WCET is a key issue in validation of safety critical applications
- aiT
  - Based on abstract interpretation
  - WCET estimation mainly based on *pipeline analysis* modelling the processor pipeline and system controllers
  - Today: Pipeline models are hand-crafted
    $\Longrightarrow$ time consuming and error-prone process
- Modern processors are derived from formal hardware descriptions
- Why not derive the pipeline analysis from the hardware description of a processor?

Problems

- [Availability/Accessibility of hardware specification]
- Processor specification too large to be used in aiT
- Specification needs to be abstracted

Idea

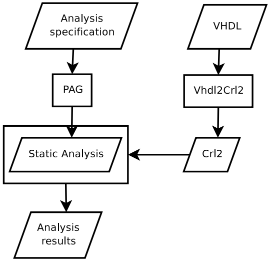
- Use of static methods to derive an abstracted model that is suitable for use in aiT
- Use of PAG as powerful generator for static analyzers

In this talk

- Framework for static analysis of VHDL code

Structure

- VHDL model is transformed into CRL2 description
  - semantically equivalent
  - some syntactical modifications (e.g. `switch` statements are transformed into `if-then-else` statements)
- Analyzer based on PAG specification
- PAG generates code working on CRL2 description



CRL2

- Provides a textual description of a control flow graph
- Hierarchically organized in
  - Routines,
  - Basic blocks,
  - Instructions and
  - Edges
- Extendible using an attribute-value concept

VHDL

- Hardware description language
- Hierarchically organized
- Defined in the IEEE standard 1076
- Focus on
  - register-transfer-level (RTL)
  - synthesizable IEEE substandard 1076.6

```
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;
```

VHDL semantics

- Two-level semantics
  - Process execution
  - Synchronization + Restart + Time

Process execution

- Sequential, imperative semantics
- Assignments to signals are delayed
- Executes, until suspended (by *wait* statement)

Second level

- After all processes have suspended
- Check if restart of processes is necessary
  - **Yes:** restart these processes (*delta cycle*)
  - **No:** wait for timeout (*theta cycle*)
- Repeat

Transformed semantics

- Ordering of process execution is not important
    - Variables are local
    - Signal assignments take effect only at synchronization point

Transform two-level semantics to one level

- Always execute all processes in fixed ordered loop
- Signal assignments can be viewed as assignments to new variables (copied at synchronization point)
    - assignment: s`<='1';` $\Longrightarrow$ s_{new}`:='1';`
    - at sync: s`:=`s_{new}`;`
- Add a *guard* to process header to check, if reexecution in the next loop iteration is necessary
    - *Guard* true, iff process is restarted at synchronization of previous iteration

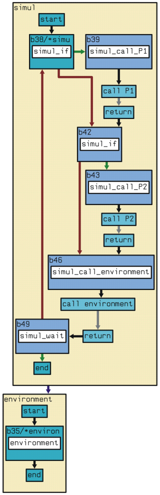
Result

Reducing two-level semantics to one level transforms the data-dependencies between processes into control-dependencies

Analysis framework

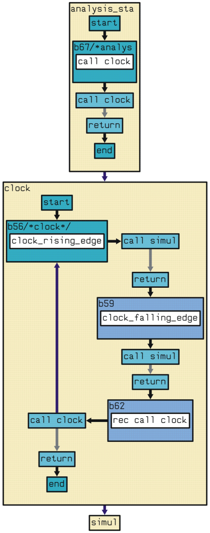
Simulation routine

- Sequential execution of processes modelled by *simul* routine
  - "Process execution" is guarded by the *simul_if* modelling the sensitivity list of the process
  - Analyzer decides, whether the edge to the call has to be taken or not
- Synchronization point is represented by *simul_wait*
- *environment* routine allows analysis of open designs



Analyzing synchronous designs

- Clock events has to be modelled separately
- Introduced special *clock* routine signalizing rising or falling events via special attributes
- Suppress uninteresting events, e.g. Leon 2 SPARC V8 implementation completely triggered on rising clock edges
- Support for multiple clock domains



Conclusion

- Framework eases the task of writing analyzers for VHDL
  - Flexible and easy to extend due to the use of PAG
- Allows analysis of open and closed designs
  - *environment* routine allows handling of open signals
- Support for multiple clock domains

Questions are guaranteed in life;
Answers aren't!

Abstractions in VHDL

1. Dead-Code Elimination
    - Slice all parts being unreachable under a specified assumption away (e.g. *reset*-signal is always *'1'*, value of a signal is within a specific range)
    - Decreases the size of the model
2. Process Substitution
    - Replace a process with an abstract process
        - Semantic of the abstract process specified in an arbitrary language (e.g. *C*)
    - Changing of domains necessary
        - Transforming datatypes (e.g. *addresses* to *address intervals*)
3. Memory Abstraction
    - Remove the memory from the VHDL model
    - Introduce new interface
        - Necessary to insert instructions into the model
        - Can be done by inserting abstract processes
    - Increases the size of the model

Generating a Timing Analysis

- Abstract the memory
    - Introduce interface to insert instruction into the model
- Find constraint: "When does an instruction leave the pipeline?"
    - Identify point in the model, where instructions complete
    - After passing this point, the completed instruction does not have any effect on signals, etc.
- Compute a backward slice for this constraint
    - All parts being not part of this slice have no effect on the timing

Generating a Timing Analysis (cont.)

- Iterate until model is handable
  - Generate code for the model
  - Simulate the resulting model (using aiT)
    - Check for state explosions and
    - Check state differences
  - Substitute a process with an abstract one
    - e.g. cache abstraction
  - Eliminate dead code