Information Flow Analysis for VHDL

Terkel K. Tolstrup, Flemming Nielson, and Hanne Riis Nielson

Informatics and Mathematical Modelling, Technical University of Denmark
{tkt, nielson, hrn}@imm.dtu.dk

Abstract. We describe a fragment of the hardware description language VHDL that is suitable for implementing the *Advanced Encryption Standard* algorithm. We then define an Information Flow analysis as required by the international standard Common Criteria. The goal of the analysis is to identify the entire information flow through the VHDL program. The result of the analysis is presented as a non-transitive directed graph that connects those nodes (representing either variables or signals) where an information flow might occur. We compare our approach to that of Kemmerer and conclude that our approach yields more precise results.

1 Introduction

Modern technical equipment often depends on the reliable performance of embedded systems. The present work is part of an ongoing effort to validate the security properties of such systems. Here it is a key requirement that the programs maintain the confidentiality of information it handles. To document this, an evaluation against the criteria of the international standard *Common Criteria* [13] is a main objective.

In this paper we focus on the Covert Channel analysis described in Chapter 14 of [13]. The main technical ingredient of the analysis is to provide a description of the direct and indirect flows of information that might occur. This is then followed by a further step where the designer argues that all information flows are permissible — or where an independent code evaluator asks for further clarification. We present the result of the analysis as a directed graph: the nodes represent the resources, and there is a direct edge from one node to another whenever there might be a direct or indirect information flow from one to the other. In general, the graph will be non-transitive [4,14].

The programming language used is the *hardware description language* VHDL [7]. Systems consist of a number of processes running in parallel where each process has its own local data space and communication between processes is performed at synchronization points using *signals*. In Section 2 we give an overview of the fragment VHDL$_1$. We present a formal Semantics of VHDL$_1$ in Section 3.

The problem of analysing VHDL programs has already been addressed in previously published approaches. The paper by Hymans [6] uses abstract interpretation to give an over-approximation of the set of reachable configurations for a fragment of VHDL not unlike ours. This suffices for checking safety properties: if the safety property is true on all states in the over-approximation it will be true for all executions of the VHDL program. Hence when synthesizing the VHDL specification one does not need to generate circuits for enforcing the reference monitor (called an observer in [6]).

The paper by Hsieh and Levitan [5] considers a similar fragment of VHDL and is concerned with optimising the synthesis process by avoiding the generation of circuits needed to store values of signals. One component of the required analyses is a Reaching Definitions analysis with a similar scope to ours although specified in a rather different manner. Comparing the precision of their approach (to the extent actually explained in the paper) with ours, we believe that our analysis is more precise in that it allows also to kill signals being set in other processes than where they are used. Furthermore the presented analysis is only correct for processes with one synchronization point, because definition sets are only influenced by definitions in other processes at the end (or beginning) of a process. Therefore definitions is lost if they are present at a synchronization point within the process but overwritten before the end of the process.

Our approach is based around adapting a Reaching Definitions analysis (along the lines of [9]) to the setting of VHDL_1. A novel feature of our analysis is that it has two components for tracking the flow of values of active signals: one is the traditional over-approximation whereas the other is an under-approximation. Finally, a Reaching Definitions analysis tracks the flow of variables and present values of signals. The details are developed in Section 4.

The first step of the Information Flow analysis determines the local dependencies for each statement; this takes the form of an inference system that is local to each process. The second step constructs the directed graph by performing the necessary “transitive closure”; this takes the form of a constraint system and makes use of the Reaching Definitions analysis. The results obtained are therefore more precise than those obtained by more standard methods like that of Kemmerer [8] and only ignore issues like timing and power-consumption. The analysis is presented in Section 5 and has been implemented in the Succinct Solver Version 1.0 [10,11] and has been used to validate several programs for implementing the NSA *Advanced Encryption Standard* (AES) [17].

2 Background

VHDL_1 is a fragment of VHDL that concentrates on the behavioral specification of models. A program in VHDL_1 consists of *entities* and *architectures*, uniquely identified by indexes $i_e, i_a \in Id$. An entity describes how an architecture is connected to the environment. The architectures comprise the behavioral or structural specification of the entities.

An entity specifies a set of signals referred to as ports ($prt \in Prt$), each port is represented by a signal ($s \in Sig$) used for reference in the specification of the architecture; furthermore a notion of the intended usage of the signal is specified by the keywords `in` and `out` defining if the signals value can be altered or read by the environment, and the type of the signal's value (either logical values or vectors of logical values).

$pgm \in Pgm$		programs
pgm	$::=$	$ent \mid arch \mid pgm_1\ pgm_2$
$ent \in Ent$		entities
ent	$::=$	$\texttt{entity}\ i_e\ \texttt{is port(}prt\texttt{); end}\ i_e;$
$prt \in Prt$		ports
prt	$::=$	$s:\ \texttt{in}\ type \mid s:\ \texttt{out}\ type \mid prt_1; prt_2$
$type \in Type$		types
$type$	$::=$	$\texttt{std\_logic} \mid \texttt{std\_logic\_vector(}z_1\ \texttt{downto}\ z_2\texttt{)}$
		$\mid \texttt{std\_logic\_vector(}z_1\ \texttt{to}\ z_2\texttt{)}$
$arch \in Arch$		architectures
$arch$	$::=$	$\texttt{architecture}\ i_a\ \texttt{of}\ i_e\ \texttt{is begin}\ css;\ \texttt{end}\ i_a;$
$css \in Css$		concurrent statements
css	$::=$	$s <= e \mid s(z_1\ \texttt{downto}\ z_2) <= e \mid s(z_1\ \texttt{to}\ z_2) <= e$
		$\mid i_p : \texttt{process}\ decl;\ \texttt{begin}\ ss;\ \texttt{end process}\ i_p$
		$\mid i_b : \texttt{block}\ decl;\ \texttt{begin}\ css;\ \texttt{end block}\ i_b \mid css_1 \vert css_2$
$decl \in Decl$		declarations
$decl$	$::=$	$\texttt{variable}\ x : type := e \mid \texttt{signal}\ s : type := e \mid decl_1; decl_2$
$ss \in Stmt$		statements
ss	$::=$	$\texttt{null} \mid x := e \mid x(z_1\ \texttt{downto}\ z_2) := e \mid x(z_1\ \texttt{to}\ z_2) := e \mid s <= e$
		$\mid s(z_1\ \texttt{downto}\ z_2) <= e \mid s(z_1\ \texttt{to}\ z_2) <= e \mid \texttt{wait on}\ S\ \texttt{until}\ e$
		$\mid ss_1; ss_2 \mid \texttt{if}\ e\ \texttt{then}\ ss_1\ \texttt{else}\ ss_2 \mid \texttt{while}\ e\ \texttt{do}\ ss$
$e \in Exp$		expressions
e	$::=$	$m \mid a \mid x \mid x(z_1\ \texttt{downto}\ z_2) \mid x(z_1\ \texttt{to}\ z_2) \mid s \mid s(z_1\ \texttt{downto}\ z_2)$
		$\mid s(z_1\ \texttt{to}\ z_2) \mid op_m^u\ e \mid e_1\ op_m^b\ e_2 \mid e_1\ op_a\ e_2$

Fig. 1. The subset VHDL$_1$ of VHDL

An architecture model is specified by a family of concurrent statements ($css \in Css$) running in parallel; here the index $i_p \in Id$ is a unique identifier in a finite set of process identifiers ($I_p \subseteq_{fin} Id$). Each process has a statement ($ss \in Stmt$) as body and may use logical values ($m \in LVal$), vectors of logical values (we write $a \in VVal$, where a has the form "$m_1 \ldots m_k$" where $m_i \in LVal$), local variables ($x \in Var$) as well as signals ($s \in Sig$, $S \subseteq_{fin} Sig$). When accessing variables and signals we always refer to their present value and when we assign to variables it is always the present value that is modified. However, when assigning to a signal its present value is *not modified*, rather its so-called active value is modified; this representation of signal's values, as illustrated in Figure 2, is used to take care of the physical aspect of propagating an electrical current through a system, the time consumed by the propagation is usually called a *delta-cycle*. The wait statements are synchronization points, where the active values of signals are used to determine the new present values that will be common to all processes.

Concurrent statements could also be block statements that allow local signal declarations for the use of internal communication between processes declared

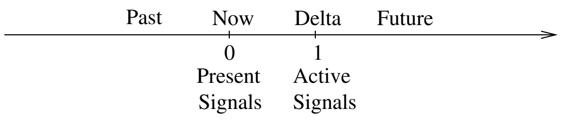

Fig. 2. The representation of abstract time in the signal store

within the block. The index $i_b \in Id$ is a unique identifier in a finite set of block identifiers ($I_b \subseteq_{fin} Id$). The scope of the local signals declared in the block definition is within the concurrent statements specified inside the block.

Signal assignment can also be performed as a concurrent statement, this corresponds to a process that is sensitive to the *free signals* in the right-hand side expression and that has the same assignment inside [2].

Since VHDL describe digital hardware we are concerned with the details of electrical signals, and it is therefore necessary to include types to represent digitally encoded values. We consider logical values ($LVal$) of the standard logic type *std_logic*, that includes traditional boolean values as well as values for electrical properties. VHDL$_1$ also allow the usage of vectors of logical values, values of this type is written using double quotes (e.g. "1" $\neq$ '1'). There are a number of arithmetic operators available on vectors of logical values.

The formal syntax is given in Figure 1. In VHDL it is allowed to omit components of wait statements. Writing $FS(e)$ for the free signals in e, the effect of '`on` $FS(e)$' may be obtained by omitting the '`on` S' component, and the effect of '`until true`' may be obtained by omitting the '`until` e' component. (In other words, the default values of S and e are $FS(e)$ and `true`, respectively.) Semantically, S is the set of signals waited on, i.e. at least one of the signals of S must have a new active value, and e is a condition on the new present values that must be fulfilled, in order to leave the wait statement.

In VHDL$_1$ the notion of signals is simplified with respect to full VHDL and thus does not allow references further into time than the following *delta-cycle*. This not only simplifies the analysis but also simplifies defining the semantics: Of the many accounts to be found in the literature [3,16] we have found the one of [16] to best correspond to our practical experiments, based on test programs simulated with the ModelSim SE 5.7d VHDL simulator. Even with this restriction VHDL$_1$ is sufficiently expressive to deal with the programs of the AES implementation.

3 Structural Operational Semantics

The main idea when defining the semantics for VHDL$_1$ programs is to execute each process by itself until a synchronization point is reached (i.e. a wait statement). When all processes of the program have reached a synchronization point synchronization is handled, while taking care of the resolution of signals in case a signal has been assigned different values by the processes. This synchronization will leave the processes in a state where they are ready either to continue execution by themselves or wait for the next synchronization.

Basic semantic domains. The syntax of programs in VHDL$_1$ is limited to statements operating on a state of logical values. These logical values are defined as $v \in LValue = \{\text{'U'}, \text{'X'}, \text{'0'}, \text{'1'}, \text{'Z'}, \text{'W'}, \text{'L'}, \text{'H'}, \text{'-'}\}$ where the values indicate the properties

'U' Uninitialized	'X' Forcing Unknown	'0' Forcing zero
'1' Forcing one	'Z' High Impedance	'W' Weak Unknown
'L' Weak zero	'H' Weak one	'-' Don't care

these values are said to capture the behavior of an electrical system better than traditional boolean values [2].

Furthermore we have vectors of logical values $a \in AValue = LValue^*$. We have a function mapping logicals in the syntax to logical values in the semantics $\mathcal{L} : LVal \rightarrow LValue$, and vectors of logical values to their semantical equivalence $\mathcal{A} : VVal \rightarrow AValue$. The semantical values are collected in the set $Value = LValue \uplus AValue$.

Constructed semantic domains. VHDL$_1$ includes local variables and signals. The values of the local variables are stored in a local state. The local state is a mapping from variable names to logical values.

$$\sigma \in State = (Var \rightarrow Value)$$

The idea is that we have a local state for each process, keeping track of assignments to local variables encountered in the execution of the process so far.

For communication between the processes we have the signals, the values of signals are stored in local states. The processes can communicate by synchronizing the signals of their local signal state with other processes.

$$\varphi \in Signals = (Sig \rightarrow (\{0,1\} \hookrightarrow Value))$$

The value assigned to a signal is available after the following synchronization, therefore we keep the present value of a signal s in $\varphi\ s\ 0$. In $\varphi\ s\ 1$ we store the assigned value, meaning that it is available after a *delta-cycle*. Each signal state has a time line for each signal. Values in the past are not used and therefore forgotten by the semantics; in VHDL$_1$ it is not possible to assign values to signals further into the future than one delta-cycle.

All signals have a present value, so $\varphi\ s\ 0$ is defined for all s. Not all signals need to be active meaning they have a new value waiting in the following delta-cycle, thus $\varphi\ s\ 1$ need not be defined; hence we use $\{0,1\} \hookrightarrow Value$ in the definition of the signal state to indicate that it is a partial function.

The semantics handles expressions following the ideas of [12]. For expressions

$$\mathcal{E} : Expr \rightarrow (State \times Signals \hookrightarrow Value)$$

evaluates the expression. The function is defined in Table 1. Note that for signals we use the current value of the signal, i.e. $\varphi\ s\ 0$.

Table 1. Semantics of Expressions

$\mathcal{E}[\![m]\!]\langle\sigma,\varphi\rangle$	$= \mathcal{L}[\![m]\!]$	
$\mathcal{E}[\![a]\!]\langle\sigma,\varphi\rangle$	$= \mathcal{A}[\![a]\!]$	
$\mathcal{E}[\![x]\!]\langle\sigma,\varphi\rangle$	$= \sigma\ x$	
$\mathcal{E}[\![x(z_1\ \texttt{downto}\ z_2)]\!]\langle\sigma,\varphi\rangle$	$= \mathit{split}(\sigma\ x, z_1, z_2)$	
$\mathcal{E}[\![s]\!]\langle\sigma,\varphi\rangle$	$= \varphi\ s\ 0$	
$\mathcal{E}[\![s(z_1\ \texttt{downto}\ z_2)]\!]\langle\sigma,\varphi\rangle$	$= \mathit{split}(\varphi\ s\ 0, z_1, z_2)$	
$\mathcal{E}[\![op^u_m\ e]\!]\langle\sigma,\varphi\rangle$	$= \overline{op^u_m}\ v$	where $\mathcal{E}[\![e]\!]\varphi = v$
		and $\overline{op^u_m}\ v$ defined
$\mathcal{E}[\![e_1\ op^b_m\ e_2]\!]\langle\sigma,\varphi\rangle$	$= v_1\ \overline{op^b_m}\ v_2$	where $\mathcal{E}[\![e_1]\!]\varphi = v_1$
		and $\mathcal{E}[\![e_2]\!]\varphi = v_2$
		and $v_1\ \overline{op^b_m}\ v_2$ defined
$\mathcal{E}[\![e_1\ op_a\ e_2]\!]\langle\sigma,\varphi\rangle$	$= v_1\ \overline{op_a}\ v_2$	where $\mathcal{E}[\![e_1]\!]\varphi = v_1$
		and $\mathcal{E}[\![e_2]\!]\varphi = v_2$
		and $v_1\ \overline{op_a}\ v_2$ defined

In the specification of the Semantics all vector values and definitions are normalized to the direction of ranging from a smaller index to a larger index. This simplification allows us to consider a significantly smaller number of rules. We define the function *split* which withdraws the elements of a vector in the range specified by the last two parameters ($\mathit{split} : a \times z \times z \rightarrow a$).

3.1 Statements

The semantics of statements and concurrent statements are specified by transition systems, more precisely by structural operational semantics. For statements we shall use configurations of the form:

$$\langle ss', \sigma, \varphi\rangle \in \mathit{Stmt}' \times \mathit{State} \times \mathit{Signals}$$

Here *Stmt′* refers to the statements from the syntactical category *Stmt* with an additional statement (`final`) indicating that a final configuration has been reached. Therefore the transition relation for statements has the form:

$$\langle ss, \sigma, \varphi\rangle \Rightarrow \langle ss', \sigma', \varphi'\rangle$$

which specifies one step of computation. The transition relation is specified in Table 2 and briefly commented upon below.

An assignment to a signal is defined as an update to the value at the delta-time, i.e. $\varphi\ s\ 1$. We use the notation $\varphi^{[i]}[s \mapsto v]$ to mean $\varphi[s \mapsto \varphi(s)[i \mapsto v]]$. For updating the variable and signal store with vector values we use the notation $\sigma[x(z_i \ldots z_j) \rightarrowtail v]$ to mean $\sigma[x \mapsto \sigma(x)[z_i \mapsto v_1] \ldots [z_j \mapsto v_{j-i}]]$, similarly for signals.

The wait statement is handled in Section 3.2, along with the handling of the concurrent processes. This is due to the fact that the wait statement is in fact a synchronization point of the processes.

Table 2. Statements

[Local Variable Assignment] :

$\langle x := e, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma[x \mapsto v], \varphi\rangle$ where $\mathcal{E}[\![e]\!]\langle\sigma,\varphi\rangle = v$

$\langle x(z_1 \texttt{ downto } z_2) := e, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma[x(z_1 \ldots z_2) \rightarrowtail v], \varphi\rangle$
where $\mathcal{E}[\![e]\!]\langle\sigma,\varphi\rangle = v$

[Signal Assignment] :

$\langle s <= e, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma, \varphi^{[1]}[s \mapsto v]\rangle$ where $\mathcal{E}[\![e]\!]\langle\sigma,\varphi\rangle = v$

$\langle s(z_1 \texttt{ downto } z_2) <= e, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma, \varphi^{[1]}[s(z_1 \ldots z_2) \rightarrowtail v]\rangle$
where $\mathcal{E}[\![e]\!]\langle\sigma,\varphi\rangle = v$

[Skip] :

$\langle \texttt{null}, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma, \varphi\rangle$

[Composition] :

$$\frac{\langle ss_1, \sigma, \varphi\rangle \Rightarrow \langle ss_1', \sigma', \varphi'\rangle}{\langle ss_1; ss_2, \sigma, \varphi\rangle \Rightarrow \langle ss_1'; ss_2, \sigma', \varphi'\rangle} \text{ where } ss_1' \in \mathit{Stmt}$$

$$\frac{\langle ss_1, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}{\langle ss_1; ss_2, \sigma, \varphi\rangle \Rightarrow \langle ss_2, \sigma', \varphi'\rangle}$$

[Conditional] :

$\langle \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2, \varphi\rangle \Rightarrow \langle ss_1, \sigma, \varphi\rangle$ if $\mathcal{E}[\![e]\!]\langle\sigma,\varphi\rangle =$ '1'
$\langle \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2, \varphi\rangle \Rightarrow \langle ss_2, \sigma, \varphi\rangle$ if $\mathcal{E}[\![e]\!]\langle\sigma,\varphi\rangle =$ '0'

[Loop] :

$\langle \texttt{while } e \texttt{ do } ss, \sigma, \varphi\rangle \Rightarrow \langle \texttt{if } e \texttt{ then } (ss; \texttt{while } e \texttt{ do } ss) \texttt{ else null}, \sigma, \varphi\rangle$

3.2 Concurrent Statements

The semantics for concurrent statements handles the concurrent processes and their synchronizations of a VHDL$_1$ program. We rewrite process declarations into statements so the process declaration i: `process` $decl_i$; `begin` ss_i; `end process` i is rewritten to `null`; `while` '1' `do` ss_i as the intention is that the statement ss is repeated indefinitely.

The transition system for concurrent statements has configurations of the form:

$$\|_{i\in I}\ \langle ss_i', \sigma_i, \varphi_i\rangle$$

for $I \subseteq_{fin} \mathit{Id}$ and $ss_i' \in \mathit{Stmt}'$, $\sigma_i \in \mathit{State}$, $\varphi_i \in \mathit{Signals}$ for all $i \in \mathit{Id}$. Thus each process has a local variable and signal state.

The initial configuration of a VHDL$_1$ program is:

$$\|_{i\in I}\ \langle \texttt{null}; \texttt{while true do } ss_i, \sigma_i^0, \varphi_i^0\rangle$$

The i^{th} process uses an initial state for signals defined by the Semantics for declarations of signals. If no initial value is specified the following are used:

$\sigma_i^0\ x =$ 'U' and $\varphi_i^0\ s\ 0 =$ 'U' for all non-vector signals used in the process ss_i. All vectors has a string of 'U''s corresponding to the length of the vector (i.e. "U...U"). $\varphi_i^0\ s\ 1$ is *undef* for all signals used in the process ss_i.

Table 3. Concurrent statements

[**Handle non-waiting processes (H)**] :

$$\frac{\langle ss_j, \sigma_j, \varphi_j\rangle \Rightarrow \langle ss'_j, \sigma'_j, \varphi'_j\rangle}{\|_{i\in I\cup\{j\}}\ \langle ss_i, \sigma_i, \varphi_i\rangle \Longrightarrow \|_{i\in I\cup\{j\}}\ \langle ss'_i, \sigma'_i, \varphi'_i\rangle}$$

where $ss'_i = ss_i \wedge \sigma'_i = \sigma_i \wedge \varphi'_i = \varphi_i$ for all $i \neq j$.

[**Active signals (A)**] :

$$\|_{i\in I}\ \langle \texttt{wait on } S_i \texttt{ until } e_i; ss_i, \sigma_i, \varphi_i\rangle \Longrightarrow \|_{i\in I}\ \langle ss'_i, \sigma_i, \varphi'_i\rangle$$

if $\exists i \in I.\ \mathit{active}(\varphi_i)$
where

$$\varphi'_i\ s\ 0 = \begin{cases} f_s\{\{v_j | \varphi_j\ s\ 1 = v_j\}\} & \text{if } \exists j \in I.\ \varphi_j\ s\ 1 \text{ is defined} \\ \varphi_i\ s\ 0 & \text{otherwise} \end{cases}$$

$$\varphi'_i\ s\ 1 = \mathit{undef}$$

$$ss'_i = \begin{cases} ss_i & \text{if } ((\exists s \in S_i.\ \varphi_i\ s\ 0 \neq \varphi'_i\ s\ 0)\wedge \\ & \quad \mathcal{E}[\![e_i]\!]\langle\sigma'_i, \varphi'_i\rangle =' 1') \\ \texttt{wait on } S_i \texttt{ until } b_i; ss_i & \text{otherwise} \end{cases}$$

The transition relation for concurrent statements has the form:

$$\|_{i\in I}\ \langle ss'_i, \sigma, \varphi_i\rangle \Longrightarrow \|_{i\in I}\ \langle ss_i{}'', \sigma'_i, \varphi'_i\rangle$$

which specifies one step of computation.

The transition relation is specified in Table 3 and explained below.

As mentioned the idea when defining the semantics of programs in VHDL$_1$ is that we execute processes locally until they have all arrived at a wait statement, this is reflected in the rule [**Handle non-waiting processes (H)**].

When all processes are ready to execute a wait statement we perform a synchronization covered by the rule [**Active signals (A)**]. If one signal waited for is active, those processes waiting for that signal may proceed; this is expressed using the predicate $\mathit{active}(\varphi)$ defined by

$$\mathit{active}(\varphi) \equiv \exists s \exists v : \varphi\ s\ 1 = v$$

The delta-time values of signals will be synchronized for all processes and in order to do this we use a resolution function $f_s : \mathit{multiset}(\mathit{Value}) \to \mathit{Value}$. Thus f_s combines the *multi-set* of values assigned to a signal into one value that then will be the new (unique) value of the signal.

Notice that even though a signal that a wait statement is waiting for becomes active, it is not enough to guarantee that it proceeds with its execution. This is because we have the side condition '`until` e'. This is reflected in the definition of the statement ss'_i of the next configuration. Notice that the state of local variables is unchanged.

3.3 Architectures

The Semantics for architectures basically initializes the local variable and signal stores for each process and rewrites the other constructions to processes. Concurrent assignments are rewritten to processes and blocks are handled by adding the signals the block declares to the scope of the processes declared inside the block. Vector variables or signals declared using the *to* specifier, where the value is reversed to match the expected ordering in the Semantics of expressions and statements.

4 Reaching Definitions Analysis

The main purpose of the Reaching Definitions analysis is to gather information about which assignments **may** have been made and not overwritten, when the execution reaches each point in the program.

The semantics divides signal states into two parts, namely the present value of a signal and the active value of a signal. Following this the analysis is divided into two parts as well, one for the active value of a signal and one for local variables and the present value of a signal. The two parts are connected since the active values of a signal influence the present value of the signal after the following synchronization. Therefore we will first define the analysis of active signals in Section 4.1, and then, that of the local variables and present values of signals in Section 4.2.

The analysis for active signals is concerned only with a single process, and thus has no information about the other processes. It collects information about which signals **might** be active in order to gather all the influences on the present value; this information is gathered for the process i by an over-approximation analysis of the active signals $RD_{\varphi}^{\cup\ i}$. It also collects information about which signals **must** be active so that the overwritten signals can be removed from the analysis result; this information is gathered for the process i by an under-approximation analysis of the active signals $RD_{\varphi}^{\cap\ i}$.

The analysis of the local variables and present values of signals will be an over-approximation. It is concerned with the entire program and thus collects information for all processes at the same time.

Common analysis domains. The analyses use a labeling scheme, a block definition and a flow relation, similar to the ones described in [9], the only difference being the wait statements which are given labels and treated as blocks. For each process i in a program $\|_{i\in I}\ i : \texttt{process}\ decl_i;\ \texttt{begin}\ ss_i;\ \texttt{end process}\ i$ the set of blocks is denoted $blocks(ss_i)$ and the flow relation is denoted $flow(ss_i)$. Similarly we use $init(ss_i)$ to denote the label of the initial block when executing the process i.

We define the cross flow relation cf for a program as the set of all possible synchronizations, i.e. cf is the Cartesian product of the set of labels of wait statements in each process.

The labeling scheme is defined so that each block has a label which is initially unique for the program. During execution the labels might not be unique within the processes, but the same label is not found in two different processes. Hence, we shall sometimes implicitly use that to each label ($l \in Lab$) there is a unique process identifier ($i \in Id$) in which it occurs.

The analyses are presented in a simplified way, following the tradition of the literature (see [9]), where all programs considered are assumed to have so-called isolated entries (meaning that the entry nodes cannot be reentered once left). This is reasonable as each process in VHDL$_1$ can be considered as a skip statement followed by a loop with an always true condition around the statement defining the process. We shall write $FV(ss)$ for the set of free variables of the statement ss and similarly $FS(ss)$ for the set of free signals.

4.1 Analysis of Active Signals

The Reaching Definitions analysis takes the form of a Monotone Framework as given in [9]. It is a *forward* Data Flow analysis, with both an over- $(RD_{\varphi}^{\cup\ i})$ and an under- $(RD_{\varphi}^{\cap\ i})$ approximation part; it operates over a complete lattice $\mathcal{P}(Sig_\star \times Lab_\star)$ where $Sig_\star$ is the set of signals and $Lab_\star$ is the set of labels present in the program.

In both cases we shall introduce functions recording the required information at the *entry* and at the *exit* of the program points. So for the over-approximation we have

$$RD_{\varphi entry}^{\cup\ i}, RD_{\varphi exit}^{\cup\ i} : Lab_\star \to \mathcal{P}(Sig_\star \times Lab_\star)$$

and similarly for the under-approximation

$$RD_{\varphi entry}^{\cap\ i}, RD_{\varphi exit}^{\cap\ i} : Lab_\star \to \mathcal{P}(Sig_\star \times Lab_\star)$$

To define the analysis we define in Table 4 a function

$$kill_{RD\varphi}^{i} : Blocks_\star \to \mathcal{P}(Sig_\star \times Lab_\star)$$

which produces a set of pairs of signals and labels corresponding to the assignments that are killed by the block. A signal assignment can be killed for two reasons: Another block in the same process assigns a new value to the already active signal, or a wait statement in the same process will synchronize all active signals, and therefore kill all signal assignments.

In Table 4 we also define the function

$$gen_{RD\varphi}^{i} : Blocks_\star \to \mathcal{P}(Sig_\star \times Lab_\star)$$

that produces a set of pairs of signals and labels corresponding to the assignments generated by the block.

The over-approximation part of the analysis is defined in terms of the information that *may* be available at the entry of the statement. Therefore the over-approximation part considers a union of the information available at the exit of all statements that have a flow directly to the statement considered.

The under-approximation part of the analysis is defined in terms of the information that *must* be available at the entry of the statement. Therefore the under-approximation part considers an intersection of the information available at the exit of all statements that have a flow directly to the statement considered.

The full details are presented in Table 4.

Table 4. Reaching Definitions Analysis for active signals; labels l are implicitly assumed to occur in process i : `process` $decl_i$; `begin` ss_i; `end process` i

kill and gen functions for the process i : `process` $decl_i$; `begin` ss_i; `end process` i
$kill^i_{RD\varphi}([s <= e]^l) = \{(s,l') \mid B^{l'} \text{ assigns to } s \text{ in process } i\}$
$kill^i_{RD\varphi}([\texttt{wait on } S \texttt{ until } e]^l) = \{(s,l') \mid B^{l'} \text{ assigns to } s \text{ in process } i\}$
$kill^i_{RD\varphi}([\ldots]^l) = \emptyset \text{ otherwise}$
$gen^i_{RD\varphi}([s <= e]^l) = \{(s,l)\}$
$gen^i_{RD\varphi}([s(z_1 \texttt{ downto } z_2) <= e]^l) = \{(s,l)\}$
$gen^i_{RD\varphi}([s(z_1 \texttt{ to } z_2) <= e]^l) = \{(s,l)\}$
$gen^i_{RD\varphi}([\ldots]^l) = \emptyset \text{ otherwise}$
data flow equations: RD_φ for the process i : `process` $decl_i$; `begin` ss_i; `end process` i
$RD^{\cup\ i}_{\varphi entry}(l) = \begin{cases} \emptyset & \text{if } l = init(ss_i) \\ \bigcup\{RD^{\cup\ i}_{\varphi exit}(l') \mid (l',l) \in flow(ss_i)\} & \text{otherwise} \end{cases}$
$RD^{\cup\ i}_{\varphi exit}(l) = (RD^{\cup\ i}_{\varphi entry}(l) \backslash kill^i_{RD\varphi}(B^l)) \cup gen^i_{RD\varphi}(B^l)$
$RD^{\cap\ i}_{\varphi entry}(l) = \begin{cases} \emptyset & \text{if } l = init(ss_i) \\ \dot{\bigcap}\{RD^{\cap\ i}_{\varphi exit}(l') \mid (l',l) \in flow(ss_i)\} & \text{otherwise} \end{cases}$
$RD^{\cap\ i}_{\varphi exit}(l) = (RD^{\cap\ i}_{\varphi entry}(l) \backslash kill^i_{RD\varphi}(B^l)) \cup gen^i_{RD\varphi}(B^l)$

For the under-approximation analysis we define a special intersection operator; $\dot{\bigcap}\emptyset = \emptyset$, and $\dot{\bigcap}X = \bigcap X$ for $X \neq \emptyset$, to guarantee that $RD^{\cap\ i}_{\varphi entry} \subseteq RD^{\cup\ i}_{\varphi entry}$, will hold for the smallest solution to the equation systems.

4.2 Analysis of Local Variables and Present Values of Signals

The Reaching Definitions analysis for the local variables corresponds to the Reaching Definitions analysis given in [9]. For the present value of signals it will use the result of the Reaching Definitions analysis for active signals. The idea is that if a signal has an active value when execution of the program arrives at a synchronization point, then the active value of the signal will become the present value of the signal after the synchronization.

The result of the Reaching Definitions analysis for active signals can be computed *before* we perform the Reaching Definitions analysis for local variables and signals. Hence the result can be considered a static set, and therefore the Reaching Definitions analysis for local variables and signals remains an instance of a Monotone Framework.

The Reaching Definitions analysis for present values of signals operates over the complete lattice $\mathcal{P}(Sig_\star \times Lab_\star)$ and is a *forward* data flow analysis. It yields an over-approximation of the assignments that **might** have influenced the present value of the signal. Its goal is to define two functions holding the information at the *entry* and *exit* of a given label in the program:

Table 5. Reaching Definitions Analysis for the local variables and present value of signals, for all labels l in the program $\|_{i\in I}\ i : \texttt{process}\ decl_i;\ \texttt{begin}\ ss_i;\ \texttt{end process}\ i$

kill and *gen* functions

$$
\begin{aligned}
kill^{cf}_{RD}([x := e]^l) &= \{(x,?)\}\cup \\
&\quad\ \{(x,l')|B^{l'} \text{ assigns to } x \text{ in process } i\} \\
kill^{cf}_{RD}([\texttt{wait on } S \texttt{ until } e]^l) &= \dot{\bigcap}_{(l_1,\ldots,l_n)\in cf, s.t.\ l_i=l} \\
&\qquad \bigcup_{j=1}^{n} fst(RD^{\cap\ i}_{\varphi entry}(l_j)) \times wS(ss_i) \\
kill^{cf}_{RD}([\ldots]^l) &= \emptyset \text{ otherwise}
\end{aligned}
$$

$$
\begin{aligned}
gen^{cf}_{RD}([x := e]^l) &= \{(x,l)\} \\
gen^{cf}_{RD}([x(z_1 \texttt{ downto } z_2) := e]^l) &= \{(x,l)\} \\
gen^{cf}_{RD}([x(z_1 \texttt{ to } z_2) := e]^l) &= \{(x,l)\} \\
gen^{cf}_{RD}([\texttt{wait on } S \texttt{ until } e]^l) &= \bigcup_{(l_1,\ldots,l_n)\in cf, s.t.\ l_i=l} \\
&\qquad \bigcup_{j=1}^{n} fst(RD^{\cup\ i}_{\varphi entry}(l_j)) \times \{l\} \\
gen^{cf}_{RD}([\ldots]^l) &= \emptyset \text{ otherwise}
\end{aligned}
$$

data flow equations: RD

$$
RD^{cf}_{entry}(l) = \begin{cases} \{(x,?) \mid x \in FV(ss_i)\} \cup \{(s,?) \mid s \in FS(ss_i)\} & \text{if } l = init(ss_i) \\ \bigcup\{RD^{cf}_{exit}(l') | (l',l) \in flow(ss_i)\} & \text{otherwise} \end{cases}
$$
$$\text{where } B \text{ and } i \text{ is uniquely given by } B^l \in blocks(ss_i)$$

$$
RD^{cf}_{exit}(l) = RD^{cf}_{entry}(l) \backslash kill^{cf}_{RD}(B^l) \cup gen^{cf}_{RD}(B^l)
$$

$$
RD^{cf}_{entry}, RD^{cf}_{exit} : Lab_\star \rightarrow \mathcal{P}((Var_\star \cup Sig_\star) \times Lab_\star)
$$

The Reaching Definitions analysis for local variables and signals is given in Table 5 and makes use of two auxiliary functions. One is

$$
kill^{cf}_{RD} : Blocks_\star \rightarrow \mathcal{P}((Var_\star \cup Sig_\star) \times Lab_\star)
$$

that produces a set of pairs of variables or signals and labels corresponding to assignments that are overwritten by the block. An assignment to a local variable will overwrite all previous assignments on the execution path. A signal value can only be overwritten by a wait statement where at least one of the synchronizing processes has an active value for the signal. To guarantee that an active value for a signal is available, the under-approximation analysis ($RD^{\cap\ i}_{\varphi}$) described above in Section 4.1 is used.

Since the active signal has to be present in all possible processes the considered wait statement could synchronize with, an intersection over the set cf of cross flow information is needed.

The other auxiliary function is

$$
gen^{cf}_{RD} : Blocks_\star \rightarrow \mathcal{P}((Var_\star \cup Sig_\star) \times Lab_\star)
$$

that produces a set of pairs of variables or signals, and labels corresponding to the assignments generated by the block. Only assignments to local variables generate definitions of a variable. Only wait statements are capable of changing a signal's value at present time. This means that in our analysis signals will get their present value at wait statements in the processes. The result of the over-approximation analysis ($RD_{\varphi}^{\cup\ i}$) contains all the signals that might be active and thus defines the present value after the synchronization. Therefore we perform a union over all the signals that might be active in any process, that might be synchronized with.

Finally, all signals are considered to have an initial value in VHDL$_1$ hence a special label (?) is introduced to indicate that the initial value might be the one defining a signal at present time. The operator *fst* is defined by $\mathit{fst}(D) = \{s \mid (s, l) \in D\}$ and extracts the first components of pairs.

5 Information Flow Analysis

The Information Flow analysis is performed in two steps. First we identify the flow of information to a variable or signal locally at each assignment; this is specified in Section 5.1. Then we perform a transitive closure of this information guided by our Reaching Definitions analysis; this is described in Section 5.2 where we also compare the result of our method with that of Kemmerer [8].

The result of the Information Flow analysis is given in the form of a directed graph. The graph has a node for each variable or signal used in the program, and an edge from the node n_1 to the node n_2 if information **might** flow from n_1 to n_2 in the program. This graph will in general be non-transitive. To illustrate this point consider the following programs:

$$\textit{(a)}: \ [\mathtt{c} := \mathtt{b}]^1; [\mathtt{b} := \mathtt{a}]^2 \qquad\qquad \textit{(b)}: \ [\mathtt{b} := \mathtt{a}]^1; [\mathtt{c} := \mathtt{b}]^2$$

In program (a) there is a flow from `b` to `c` and a flow from `a` to `b` and therefore the resulting graph shown in Figure 3(a) has an edge from node `b` to node `c` and an edge from node `a` to node *b*. There is no flow from `a` to `c` and indeed there is no edge from `a` to `c`. In program (b) on the other hand there is a flow from `a` to `c` and the resulting graph shown in Figure 3(b) indeed has an edge from `a` to `c`.

5.1 Local Dependencies

It is clear that an assignment of a variable to another variable will cause a flow of information. As an example, `a` := `b` causes a flow of information from `b` to `a`. We also need to consider implicit flows due to conditional statements. As an example, `if c then a := b else null` has an implicit flow from `c` to `a` because an observer could use the resulting value of `a` to gain information about the value of `c`.

In this fashion we must consider all the statements of VHDL$_1$ and determine how information might flow. For a VHDL$_1$ program we define a set of structural rules that define the set of dependencies between local variables and signals. The analysis is defined using judgments of the form

$$B \vdash ss : RM$$

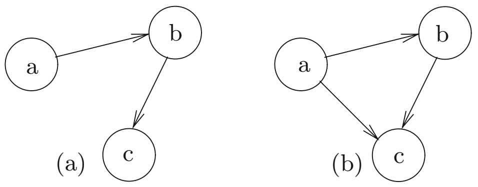

Fig. 3. Result of the Information Flow Analysis for programs (a) and (b)

where $B \subseteq (\mathbf{Var} \cup \mathbf{Sig})$, $ss \in \mathbf{Stmt}$ and $RM \subseteq ((\mathbf{Var} \cup \mathbf{Sig}) \times \mathbf{Lab} \times \{M_0, M_1, R_0, R_1\})$. Here ss is the statement analyzed under the assumption that it is only reachable when values of variables and signals in B have certain values. The result is the set RM containing entries (n, l, M_j) if the variable or signal n might be modified at label l in ss; we use M_0 for variables and present values of signals and M_1 for active values of signals. Similarly, RM contains entries (n, l, R_j) if the variable or signal n might be read at label l in ss; we use R_0 for variables and present values of signals and R_1 for the synchronization of active values of signals.

The local dependency analysis of the flow between variables and signals is specified in Table 6 and is explained below. Assignments to variables result in local dependencies, there are no other statements that causes information to flow into variables.

For the active signals in a program it holds that information can only flow to the signal through signal assignment. Hence only the signal assignment contributes dependencies to the resulting set. Notice that the information flowing to active signals (M_1) might come from both local variables and the present value of signals, but never from the active value of signals.

The variables and signals used in the evaluation of conditions within `if` and `while` statements are collected in the *block-set* B as they might implicitly flow into assigned variables or signals in the branches. This is taken care of in rules [**Conditional**] and [**Loop**]. Notice that these rules do not handle termination or timing channels that might occur.

The synchronization statements (i.e. `wait`) cause information about the active signals to flow to the present value of the same signals. Hence we will update (writing R_1) all signals present in the process considered.

5.2 Global Dependencies

Using the local dependencies defined above we can construct a Resource Matrix specifying for each point in the program which resources (i.e. a variable or signal) was modified and which resources were read meanwhile [8]. First we apply the local dependency analysis on each process in the considered program the result is collected in $RM_{lo} = \bigcup_i RM_i$ where $\emptyset \vdash ss_i : RM_i$. Then we need to compute the global dependencies; one way to do this is to take the transitive closure of

Table 6. Structural rules for constructing a Resource Matrix for the process i : `process` $decl_i$ `begin` ss_i; `end process` i

[Local Variable Assignment] :

$$B \vdash [x := e]^l : \{(x, l, M_0)\} \cup \{(n, l, R_0) \mid n \in FV(e) \cup FS(e) \cup B\}$$

$$B \vdash [x(z_1 \texttt{ downto } z_2) := e]^l : \{(x, l, M_0)\} \cup \{(n, l, R_0) \mid n \in FV(e) \cup FS(e) \cup B\}$$

$$B \vdash [x(z_1 \texttt{ to } z_2) := e]^l : \{(x, l, M_0)\} \cup \{(n, l, R_0) \mid n \in FV(e) \cup FS(e) \cup B\}$$

[Signal Assignment] :

$$B \vdash [s <= e]^l : \{(s, l, M_1)\} \cup \{(n, l, R_0) \mid n \in FV(e) \cup FS(e) \cup B\}$$

$$B \vdash [s(z_1 \texttt{ downto } z_2) <= e]^l : \{(s, l, M_1)\} \cup \{(n, l, R_0) \mid n \in FV(e) \cup FS(e) \cup B\}$$

$$B \vdash [s(z_1 \texttt{ to } z_2) <= e]^l : \{(s, l, M_1)\} \cup \{(n, l, R_0) \mid n \in FV(e) \cup FS(e) \cup B\}$$

[Skip] :

$$B \vdash [\texttt{null}]^l : \emptyset$$

[Composition] :

$$\frac{B \vdash ss_1 : RM_1 \qquad B \vdash ss_2 : RM_2}{B \vdash ss_1; ss_2 : RM_1 \cup RM_2}$$

[Conditional] :

$$\frac{B' \vdash ss_1 : RM_1 \qquad B' \vdash ss_2 : RM_2}{B \vdash \texttt{if } [e]^l \texttt{ then } ss_1 \texttt{ else } ss_2 : RM_1 \cup RM_2}$$

where $B' = B \cup FV(e) \cup FS(e)$

[Loop] :

$$\frac{B' \vdash ss : RM}{B \vdash \texttt{while } [e]^l \texttt{ do } ss : RM}$$

where $B' = B \cup FV(e) \cup FS(e)$

[Synchronization] :

$$B \vdash [\texttt{wait on } S \texttt{ until } e]^l : \{(s, l, R_1) \mid s \in FS(ss_i)\} \cup \{(n, l, R_0) \mid n \in B \cup S \cup FV(e) \cup FS(e)\}$$

where ss_i is the body of process i in which l resides

the local dependencies; this method is attributed to Kemmerer and is described in [8] in case of traditional programming languages.

Let us evaluate the traditional method for constructing the Resource Matrix. For this we consider the program (a) defined above. The result of the transitive closure will correspond to the graph presented in Figure 3(b), but not to the true behavior of the program as depicted in the graph in Figure 3(a). This is due to the flow-insensitivity of the transitive closure method: The imprecision is a result of the method failing to consider information about the flow between labels in the programs.

Table 7. Specialization of $RD^{\cup\, i}_{\varphi entry}$ and RD^{cf}_{entry}

[RD for active signals]

$$\frac{(s, l_i, R_1) \in RM_{lo} \quad (s, l) \in RD^{\cup\, i}_{\varphi entry}(l_i)}{(s, l) \in RD^{\dagger}_{\varphi}(l_i)} \qquad \text{if } \exists\, \overrightarrow{l} \in cf : l_i \text{ occurs in } \overrightarrow{l}$$

[RD for present signals and local variables]

$$\frac{(n, l', R_0) \in RM_{lo} \quad (n, l) \in RD^{cf}_{entry}(l')}{(n, l) \in RD^{\dagger}(l')}$$

Table 8. Transitive closure of Resource Matrix, based on $RD^{\dagger}$ and $RD^{\dagger}_{\varphi}$

[Initialization]

$$\frac{(n, l, A) \in RM_{lo}}{(n, l, A) \in RM_{gl}} \quad \text{where } A \in \{R_0, R_1, M_0, M_1\}$$

[Present values and local variables]

$$\frac{(n', l') \in RD^{\dagger}(l) \quad (n, l', R_0) \in RM_{gl}}{(n, l, R_0) \in RM_{gl}}$$

[Synchronized values]

$$\frac{(s', l_i) \in RD^{\dagger}(l) \quad (s', l'') \in RD^{\dagger}_{\varphi}(l_j) \quad (s, l'', R_0) \in RM_{gl}}{(s, l, R_0) \in RM_{gl}}$$
$$\text{if } \exists\, \overrightarrow{l} \in cf : l_i \text{ and } l_j \text{ occur in } \overrightarrow{l}$$

Closure based on Reaching Definitions. This motivates modifying the closure condition to make use of Reaching Definitions information. Indeed the Reaching Definitions analysis specified in Section 4 supplies us with the needed information to exclude some of the “spurious flows” when performing the transitive closure.

Before doing so we specialize in Table 7 the result of the Reaching Definitions analysis to allow a better precision in the closure of the Resource Matrix. The specialization ensures that definitions are only considered to reach a labeled construct if they are actually used in the labeled construct. This is done by considering the result of the local dependency analysis; notice the usage of the cross flow relation in rule [**RD for active signals**] which determines if the signal might in fact be synchronized.

We can now update the specification of the transitive closure using the result of the Reaching Definitions analysis, as is done in Table 8. We specify a rule for initializing the Global Resource Matrix [**Initialization**].

The closure is done by rule [**Present values and local variables**] considering the result of the Reaching Definitions analyses for the program. For the present value of a signal and for local variables we consider each entry in the Resource Matrix, if the present value of a variable or signal is read (R_0) we can use the information of the Reaching Definitions analysis to find the label where the variable or signal was defined. Therefore we copy all the entries about variables and signals read at this label in the Resource Matrix. This rule also handles the

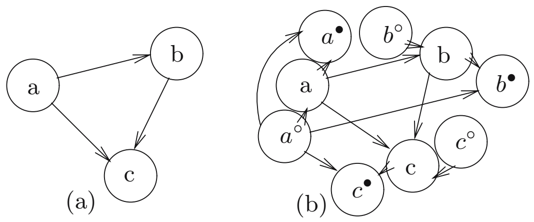

Fig. 4. Result of the Information Flow Analysis for program (b)

case where information flows from the variables and signals in a condition on a synchronization point.

The rule [**Synchronized values**] uses the result of the Reaching Definitions analysis to determine which signals were read in the Resource Matrix and follow them to their definition. When synchronizing signals the matter is complicated as the signal is defined at a synchronization point, therefore the rule needs to consider all the information about signals flowing into the synchronization points that might be synchronized with. Which synchronization points the definition point synchronizes with is gathered in the *cf* predicate, hence we apply the Reaching Definitions analysis for active signals on all the synchronization points and copy all the entries indicating variables and signals being read from the point the signal could be defined.

5.3 Improvement of the Information Flow Analysis

For the example program (b) (i.e. $[\texttt{b} := \texttt{a}]^1; [\texttt{c} := \texttt{b}]^2$) we previously described how the Information Flow Analysis would yield the result presented in Figure 4(a). In fact the resulting graph indicates that the resulting value of the variable `b` can be read from the resulting value of the variable `c`, which is entirely correct. However, the initial value of the variable `b` cannot be read from the variable c; to see this consider a scenario where `b` initially contained a value, this value would never flow to `c`, as the first assignment would overwrite the variable.

The Information Flow Analysis based on the Reaching Definitions Analysis can be improved to handle the initial and outgoing values of signals with greater accuracy. The idea is to add a node to the graph for each incoming signal, annotating the incoming node of a variable with a $\circ$, and for each outgoing signal, annotating the outgoing node with a $\bullet$. Using this scheme a more precise result for program (b) can be constructed as shown in Figure 4(b), where we consider the last statement to be outcoming and therefore update the Resource Matrix in the same fashion as for wait statements.

The extension of the analysis is based on adding special variables and signals for incoming and outgoing values. The rules for improving the information flow analysis are presented in Table 9 and explained below.

In a traditional sequential programming language the improvement could be handled by adding assignments of the form $x := x^\circ$ for each variable read in front of the program, and similarly adding assignments $x^\bullet := x$ in the end for

Table 9. Rules for the improved Information Flow Analysis

[Initial values]

$$\frac{(n,?) \in RD^{\dagger}(l)}{(n^{\circ}, l, R_0) \in RM_{gl}}$$

[Incoming values]

$$\frac{(n,l') \in RD^{\dagger}(l) \qquad l' \in WS}{(n^{\circ}, l, R_0) \in RM_{gl}}$$

[Outgoing values]

$$\frac{n \in Sig^{out}}{(n^{\bullet}, l_{n^{\bullet}}, M_1) \in RM_{gl}}$$

[Outcoming values]

$$\frac{l \in WS \qquad (n,l') \in RD^{\dagger}_{\varphi}(l) \quad (n', l', R_0) \in RM_{gl}}{(n', l_{n^{\bullet}}, R_0) \in RM_{gl}}$$

handling the outgoing values. Having this in mind we introduce the rule [**Initial values**] that uses the special symbol (?) from the Reaching Definitions analysis to propagate the initial value of a variable or locally defined signal.

VHDL$_1$ consists of processes running as infinite loops in parallel with other processes and under the influence of the environment. Therefore signals might carry incoming values at any synchronization point, similarly a process might communicate values out of the system at any synchronization point. We introduce a new process π to illustrate how the incoming and outgoing signals are handled. The process has the form

$$\pi : \texttt{process begin } [s_1^{in} <= s_1^{\circ}]; \ldots [\texttt{wait on } S^{\pi}]; [s_1^{\bullet} <= s_1^{out}]^{l_{s_1^{\bullet}}}; \ldots$$
$$\texttt{end process } \pi$$

where $s_1^{in}, s_2^{in}, \ldots$ are the incoming signals, $s_1^{out}, s_2^{out}, \ldots$ are the outgoing signals and S^{π} is the set of all incoming and outgoing signals, as specified in the entity declaration of the program.

The assignments prior to the synchronization point in process π can be synchronized into the system at each wait statement and this is handled by rule [**Incoming values**] where $WS = \bigcup_i WS(ss_i)$ are all the wait statements in the program.

We add two rules for the outgoing values to the closure method. The first rule [**Outgoing values**] specifies a special label for each signal (i.e. the label of the assignment to $n^{\bullet}$ in process π) used on the left-hand side in an assignment, at which the signal is set to be modified in the Resource Matrix. The second rule [**Outcoming values**] handles the right-hand side of the outgoing assignments in process π by considering all active signals coming into a wait statement, the values read when these active signals where modified are the signals that influence the outgoing value. Sig^{out} is the set of signals that are declared as outgoing (i.e. with the keyword `out` in the entity declaration).

6 Results

In order to compare our work to Kemmerer's method we shall consider part of the NSA *Advanced Encryption Standard* test implementation of the 128 bit version of the encryption algorithm [17]. Both the presented analyses and Kemmerer's method have been implemented using the Succinct Solver.

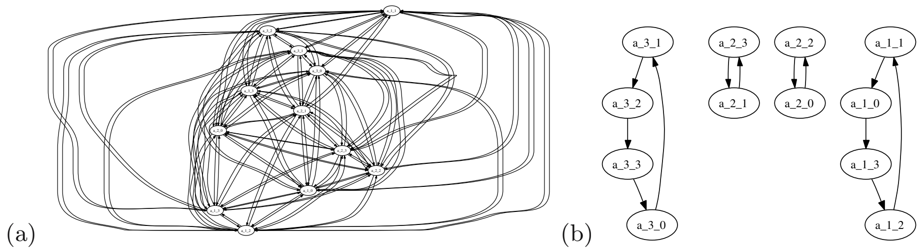

Fig. 5. Resulting graphs of Kemmerer's method (a) and our analysis (b) on a shift function

The analysed programs use several temporary variables. These variables are overwritten and reused for each input state. The graphs computed by Kemmerer's method indicate the problem of the method not taking control flow information into account; many edges are false positives resulting from the over approximation. Our analysis correctly eliminates the edges introduced by the overwritten variables.

To illustrate the difference between the two approaches, we consider the function shifting rows in a block. The first row is not altered by the function, while the last three rows are shifted 1, 2 and 3 positions respectively. The values flow through temporary variables, which are used for all three rows. The function is preprocessed by unrolling the loops and replacing constants with their values.

The resulting graphs are simplified so that only the nodes for the three shifted rows are presented. Furthermore we have merged incoming and outgoing nodes in the graph of our analysis. Therefore both of the resulting graphs for the three shifted rows have 12 nodes, and are now comparable. Kemmerer's method is unable to separate the shifts on each row as shown in Figure 5(a). Our analysis computes the precise result as shown in Figure 5(b).

7 Conclusion

One main achievement of the paper is the adaptation of the classical notion of Reaching Definitions analysis from traditional programming languages to real-time languages in the context of hardware description languages. We performed a development for a useful fragment of VHDL and correctly deal with the complications due to active and present values of signals. One unusual ingredient is the under-approximation analysis for active signals in order to be able to specify non-trivial kill-components for present values.

The other main achievement is the demonstration of the usefulness of the Reaching Definitions analysis for developing an Information Flow analysis that is more precise than the traditional method of Kemmerer. The local dependencies were specified by a straightforward inference system in the manner of information flow analyses. The global dependencies made good use of all aspects of the Reaching Definitions analysis.

Furthermore the improved information flow analysis correctly analyses programs that would incorrectly be rejected by typical security-type systems; as it is described in the *Open Challenge F* of [15]. This is due to the fact that the Reaching Definitions analysis allows us to kill overwritten variables and signals.

The current implementation directly follows the structure of the specifications given in the previous sections and one can argue that its worst case complexity is $O(n^5)$. So far this has posed no problems, however we conjecture that the implementation can be improved to have a cubic worst case complexity. The reason is that the analysis basically is a combination of three bit-vector frameworks (each being linear time in practice) [9] and a cubic time reachability analysis [1].

References

1. A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.
2. P. J. Ashenden. *The Designer's Guide To VHDL.* Morgan Kaufmann, 2nd edition, 2002.
3. K. G. W. Goossens. Reasoning About VHDL Using Operational and Observational Semantics. In *CHDM*, volume 987 of *LNCS*, pages 311–327. Springer, 1995.
4. J. T. Haigh and W. D. Young. Extending the Non-Interference Version of MLS for SAT. In *IEEE Symposium on Security and Privacy*, pages 232–239, 1986.
5. Y-W. Hsieh and S. P. Levitan. Control/Data-Flow Analysis for VHDL Semantic Extraction. *Journal of Information Science and Engineering*, 14(3):547–565, 1998.
6. C. Hymans. Checking Safety Properties of Behavioral VHDL Descriptions by Abstract Interpretation. In *SAS*, volume 2477 of *LNCS*, pages 444–460. Springer, 2002.
7. IEEE inc. *IEEE Standard VHDL Language Reference Manual.* IEEE, 1988.
8. J. McHugh. Covert Channel Analysis. *Handbook for the Computer Security Certification of Trusted Systems*, 1995.
9. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999.
10. F. Nielson, H. R. Nielson, and H. Seidl. A Succinct Solver for ALFP. *Nordic Journal of Computing*, 9(4):335–372, 2002.
11. F. Nielson, H. R. Nielson, H. Sun, M. Buchholtz, R. R. Hansen, H. Pilegaard, and H. Seidl. The Succinct Solver Suite. In *TACAS*, volume 2988 of *LNCS*, pages 251–265. Springer, 2004.
12. H. R. Nielson and F. Nielson. *Semantics with Applications - A Formal Introduction.* John Wiley & Sons, 1992.
13. International Standards Organisation. Common Criteria for information technology security (CC). *ISO/IS 15408 Final Committee Draft, version 2.0.*, 1998.
14. J. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report CSL-92-02, SRI International, December 1992.
15. A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
16. K. Thirunarayan and R. L. Ewing. Structural Operational Semantics for a Portable Subset of Behavioral VHDL-93. *FMSD*, 18(1):69–88, 2001.
17. B. Weeks, M. Bean, T. Rozylowicz, and C. Ficke. Hardware performance simulations of round 2 advanced encryption standard algorithms. Technical report, National Security Agency, 2000.