Mapping Abstract and Concrete Hardware Models for Design Understanding

Tino Flenker* and Görschwin Fey*†
*University of Bremen, Institute of Computer Science, 28359 Bremen, Germany
†Institute of Space Systems, German Aerospace Center, 28359 Bremen, Germany
Email: {flenker, fey}@informatik.uni-bremen.de

***Abstract*—Before a microchip's concrete implementation is available a very abstract model is created, e.g., on *Electronic System Level* (ESL) or even more abstract. To ensure a better design understanding, we propose an automated mapping from a given abstract model to an unfamiliar concrete implementation at *Register Transfer Level* (RTL). But how to map a variable from the abstract model to a variable from the concrete model? We address this problem by a simulation based approach. We instrument the abstract model to get traces for each variable in both models and propose four heuristics to evaluate which variable maps to a corresponding variable of the other model. Experiments on an *Instruction Set Simulator* (ISS) versus RTL processor show mappings which offer an insight into the RTL implementation.**

Keywords — Design Understanding

I. Introduction

Microchip's complexity increases at tremendous speed. To adhere to strict time-to-market constraints, tools are required to facilitate a rapid understanding and incorporation of hardware designs. This makes design understanding an important research topic. A faster understanding supports new colleagues in a company, enables a more efficient debugging tool for support, and reduces the training period.

During microchip's development a module is described on different abstraction levels. This is done by checking the microchip's functional design at an early stage on an abstract level. Later, the equivalence of different models is ensured. Over time a project increases in complexity. Old code is reused and third party libraries are used so that no one knows every detail of each model. A mapping of distinctive parts from one model to another model on a different level of abstraction is desirable. However, no accurate mapping between very abstract and concrete models can be expected, due to unavailability of structural and timing information.

To solve *Equivalence Checking* (EC) between abstraction levels, several approaches have been proposed. In [1][2] formal methods are used but it is not feasible to perform EC using conventional equivalence checkers due to significant internal differences in the very abstract and the concrete model. Simulation is used to find potentially equivalent nodes as a preprocessing step for formal EC. Such equivalent nodes are often called cut-points. This also applies to situations dealing with concrete models and relatively detailed abstract ones [3][4][5][6]. However, EC often expects complete equivalence of models including cycle accuracy [7]. These techniques are not suitable in our case, because of too many structural differences and the absence of timing information on the abstract level.

In [8] Chauhan et al. proposed a method for non cycle accurate EC. However, this approach is only suitable for RTL to RTL EC so it is not relevant for an ESL to RTL mapping. Groe et al. [9] used a simulation-based approach with the assumption that both models are available in SystemC [10]. In [11], Fujita et al. present three techniques that utilize models on ESL for debugging designs of various lower abstraction levels. They work across multiple abstraction levels but new statements, synchronization statements, and modules are introduced to describe hardware oriented behavior on abstract level.

For design understanding, we propose an approach to map very abstract models to concrete models. Our approach uses simulation to find corresponding parts of the original model in a model of another abstraction level. We aim to find characteristic parts of a more abstract model than typical ESL written in C/C++ in a concrete model available on RTL written in Verilog. By this, the designer can directly find the implementation of abstract functionality. Mapping models is reduced to mapping variables between the two abstraction levels. We use simulation traces to perform the mapping. The underlying assumption is that variables relating to the same functionality yield similar simulation traces.

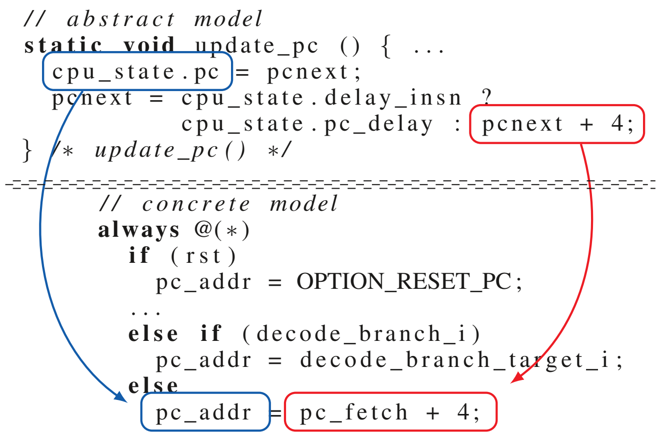
However, for design understanding it is harder to find correspondences than for EC because the two models are expected to be quite different. Usually a very abstract model lacks the following information:

- Timing
  Generally, the very abstract module implements the functional behavior in an early state of the development. Thus, cycle accurate timing is not implemented and is consequently absent.
- Structure
  Useful structural information is typically not available on such an abstract level. Only a very coarse module structure is described. For example, consider the implementation of a processor. Implementing a pipeline for an ISS is not essential because the absence does not change the functional design but the pipeline exists on RTL.

Fig. 1 shows a mapping of an abstract to a concrete processor implementation. A mapping of a *Program Counter* (PC) variable (blue) from the abstract (above) to the concrete model (below) is shown. Fig. 1 also shows the part of the source codes, where the PC is increased by one operation (red mapping). Further mappings are tough because, e.g., concepts like pipelining are not included in the abstract implementation.

This work was supported by the University of Bremens Graduate School SyDe, funded by the German Excellence Initiative, and the German Research Foundation (DFG, grant no. FE 797/6-2).

This work has also been partially funded from the European Unions Horizon 2020 research and innovation programme under grant agreement No. 644905 (IMMORTAL).



```
// abstract model
static void update_pc () { ...
  cpu_state.pc = pcnext;
  pcnext = cpu_state.delay_insn ?
           cpu_state.pc_delay : pcnext + 4;
} /* update_pc() */
```

```
      // concrete model
      always @(*)
        if (rst)
          pc_addr = OPTION_RESET_PC;
        ...
        else if (decode_branch_i)
          pc_addr = decode_branch_target_i;
        else
          pc_addr = pc_fetch + 4;
```

Figure 1: Mapping of PC from abstract to concrete model

Section II-A presents a short overview of the work flow. Four heuristics are presented which compare traces of abstract and concrete models in Section II-B. The heuristics handle traces despite lacking timing information and the different structure of the implementations. Experimental results in Section III show that a mapping of functionally similar parts is possible and Section IV offers a short conclusion.

II. Methodology

This section describes the proposed approach. First, the work flow is presented and afterwards four heuristics for variable mapping are explained.

The main goal is to find relations between variables of an abstract and a concrete hardware model. For that an implementation written in a *Hardware Description Language* (HDL) and another implementation written in a higher level programming language like C/C++ or SystemC are considered. An example for an abstract implementation is on a processor with an ISS model.

A. Work Flow

The work flow of our approach is illustrated in Fig. 2. Given are an abstract model $\mathcal{A}$ with variables $\mathcal{V}^{\mathcal{A}}$ and also a concrete model $\mathcal{C}$ with variables $\mathcal{V}^{\mathcal{C}}$. The set $\mathcal{V}al$ includes all occurring values in all traces. To get a trace of each implementation some use cases are required as stimuli, which execute the same functionality and are available for both models. For the hardware model's trace $\mathcal{C}$ is simulated and the values of registers and internal signals are printed to $\mathcal{T}^{\mathcal{C}}$. The trace of the abstract model $\mathcal{T}^{\mathcal{A}}$ is gained by the instrumentation of $\mathcal{A}$ so that the values are collected, when needed. Next, the traces are used to find a mapping between the variables of models. Section II-B presents four heuristics to compute a fitting value for a mapping between to variables. Finally, a mapping between all variables is gained, which is needed for presenting corresponding variables between both models.

B. Trace Analysis

This section provides preliminary definitions. Afterwards four heuristics for mapping variables are introduced. Given the variable's values in the traces we consider the following heuristics:

1) The symmetric difference between the sets of values.
2) The intersection between the sets of values.
3) The number of occurrences of each variable's values.
4) The sequence of variable's values.

The set of traces for all variables in the abstract model is defined as follows:

$$\mathcal{T}^{\mathcal{A}} := \{(x, s) \mid x \in \mathcal{V}^{\mathcal{A}} \text{ and } s = (v_1, \dots, v_n) \text{ with } v_i \in \mathcal{V}al\}$$

$\mathcal{T}^{\mathcal{A}}$ is a set containing pairs of a variable and an associated sequence. The variable x belongs to the abstract model and the sequence s includes all values in order of occurrence in time. No value occurs twice in succession. $\mathcal{T}^{\mathcal{A}}$ includes the traces for all variables for one use case. The set of traces for the concrete model $\mathcal{T}^{\mathcal{C}}$ is defined analogously.

For the mapping *cand* takes the cross product of $\mathcal{V}^{\mathcal{A}}$ and $\mathcal{V}^{\mathcal{C}}$ and maps each pair to an integral number, which indicates the quality of the mapping:

$$cand : \mathcal{V}^{\mathcal{A}} \times \mathcal{V}^{\mathcal{C}} \rightarrow \mathbb{Z}$$

The auxiliary function $s(\mathcal{T}, x)$ yields the set of all values assigned to variable x occurring in a given trace $\mathcal{T}$:

$$s(\mathcal{T}, x) := \{v_i \mid (x, (v_1, \dots, v_n)) \in \mathcal{T} \wedge 1 \leq i \leq n\}$$

1) Symmetric Difference: The first heuristic collects all values of each variable in a separate set. To match traces, the symmetric difference $SD(x, y)$ is taken.

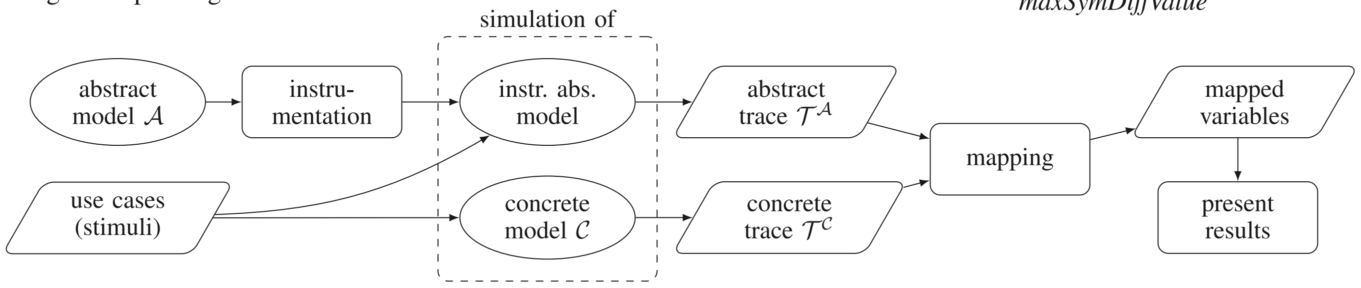
$$SD(x, y) := (s(\mathcal{T}^{\mathcal{A}}, x) \cup s(\mathcal{T}^{\mathcal{C}}, y)) \backslash (s(\mathcal{T}^{\mathcal{A}}, x) \cap s(\mathcal{T}^{\mathcal{C}}, y))$$

The variables with the smallest symmetric difference are the best fit. The variables with the best fit are considered as a candidate mapping. The function $cand_1(x, y)$ takes the variables $x \in \mathcal{V}^{\mathcal{A}}$ and $y \in \mathcal{V}^{\mathcal{C}}$ as input and returns the number of elements in the symmetric difference. Mappings with the least number of differences are considered as the best fit:

$$cand_1(x, y) := |SD(x, y)|$$

Finally, we define a normalization $unify_1(x, y)$ that becomes 1 for likely matching candidates. The result of $cand_1(x, y)$ is divided by the worst existing symmetric difference value. The result of the division is subtracted from 1. A unified value close to 1 means, that the compared variables have a small symmetric difference, which represents a good fit value.

$$unify_1(x, y) := 1 - \frac{cand_1(x, y)}{maxSymDiffValue}$$



simulation of
abstract model $\mathcal{A}$
instru-mentation
instr. abs. model
abstract trace $\mathcal{T}^{\mathcal{A}}$
mapped variables
mapping
use cases (stimuli)
concrete model $\mathcal{C}$
concrete trace $\mathcal{T}^{\mathcal{C}}$
present results

Figure 2: Work flow for proposed approach

2) Intersection: The second heuristic uses the intersection of two variable's values.

$$INS(x,y) := s(\mathcal{T}^{\mathcal{A}},x) \cap s(\mathcal{T}^{\mathcal{C}},y)$$

The variables with the largest intersection are considered as the best fit:

$$cand_2(x,y) := |INS(x,y)|$$

The value of this heuristic is unified by $\mathit{unify}_2(x,y)$. The size of the intersection of the compared variables x and y is divided by the number of values of the variable belonging to $\mathcal{T}^{\mathcal{A}}$.

$$\mathit{unify}_2(x,y) := \frac{cand_2(x,y)}{|\ s(\mathcal{T}^{\mathcal{A}},x)\ |}$$

Both heuristics assume that signals for data get the same values in both implementations. Consequently, in one hand the result is the same set of values which implies that the symmetric difference is the smallest. On the other hand, the same set of values of two variables implies that the intersection is the largest.

Hard to distinguish are control signals which typically have a bit width of one. The sets of values of these signals contain at most the values 0 and 1. Consequently, the control signals are indistinguishable using this simple heuristic.

3) Count: The third heuristic encounters the drawback of method II-B1 and II-B2. This heuristic assumes that different control signals toggle a different number of times. In this manner the control signals can be distinguished. This heuristic collects all values of the different variables, too. But in addition to that, each occurrence of a value is counted. For this purpose the function *occ* calculates how often the variable x has assigned the value v in sequence $(v_1,\ldots,v_n)$.

$$\mathit{occ}(\mathcal{T},x,v) := |\{i \mid (x,(v_1,\ldots,v_n)) \in \mathcal{T} \wedge \\ 1 \le i \le n \wedge v_i = v\}|$$

To match traces by the *count* heuristic, first the function *diff* is introduced. This function computes the difference between the number of occurrences of value v in trace $\mathcal{T}^{\mathcal{A}}$ assigned by variable x and v in trace $\mathcal{T}^{\mathcal{C}}$ assigned by variable y:

$$\mathit{diff}(x,y,v) := |\mathit{occ}(\mathcal{T}^{\mathcal{A}},x,v) - \mathit{occ}(\mathcal{T}^{\mathcal{C}},y,v)|$$

The sum of all differences between two variables is called divergence. The function $cand_3(x,y)$ computes the divergence between the given two variables $x \in \mathcal{V}^{\mathcal{A}}$ of the abstract model and $y \in \mathcal{V}^{\mathcal{C}}$ of the concrete model.

$$cand_3(x,y) := \sum_{v \in \mathcal{V}al} \mathit{diff}(x,y,v)$$

The variables with the smallest divergence are the best fit to be candidates of the mapping. This means the two corresponding variables have fewer differences in the number of occurrences of the values than two unrelated variables.

The number of occurrences on both traces may vary substantially. The reason is the difference in the timing information in the models which is almost missing in the abstract model while it is cycle accurate in the concrete implementation.

The function $\mathit{unify}_3(x,y)$ unifies the results of $cand_3(x,y)$. From 1 the quotient of $cand_3()$ and the worst existing value of the third heuristic is subtracted. A result close to 1 indicates, that the difference of the count of the values of x and y is small which represents a good matching.

$$\mathit{unify}_3(x,y) := 1 - \frac{cand_3(x,y)}{\mathit{worstCountValue}}$$

Algorithm 1 Computation of penalty value

```
1:  function penalty($s_1, s_2$)
2:      $m_1 \leftarrow$ eval($s_1$), $m_2 \leftarrow$ eval($s_2$)
3:      $p = 0,\ i_1 = 0,\ i_2 = 0$
4:      while $i_1 <$ length($s_1$) $\wedge\ i_2 <$ length($s_2$) do
5:          $v_1 \leftarrow s_1[i_1],\ v_2 \leftarrow s_2[i_2]$
6:          if $v_1 \equiv v_2$ then
7:              $i_1 \leftarrow i_1 + 1,\ i_2 \leftarrow i_2 + 1$
8:              $m_1[v_1]$.pop(), $m_2[v_2]$.pop()
9:          else
10:             $p \leftarrow p + 1$
11:             $n_1 \leftarrow m_1[v_2]$.top()
12:             $n_2 \leftarrow m_2[v_1]$.top()
13:             if $(n_1 - i_1) < (n_2 - i_2)$ then
14:                 $i_1 \leftarrow i_1 + 1$
15:                 $m_1[v_1]$.pop()
16:             else
17:                 $i_2 \leftarrow i_2 + 1$
18:                 $m_2[v_2]$.pop()
19:     end while
20:     $p \leftarrow p + (($length($s_1$) $- i_1) + ($length($s_2$) $- i_2)) \cdot w$
21:     return $p$
```

4) Sequence: The fourth heuristic considers the sequence of values to map the variables between the abstraction levels. Two sequences with equal values of data indicate two variables representing the same semantics on the functional level.

The mapping computed by this heuristic is as follows. Two sequences are observed value by value. If two values are not equal, then all positions up to the next identical value are skipped. Each skipped value increases the penalty by one.

The function $\mathit{seq}(\mathcal{T},x)$ gets the sequence of x out of $\mathcal{T}$:

$$\mathit{seq}(\mathcal{T},x) := s \quad \text{with} \quad (x,s) \in \mathcal{T}$$

The function $cand_4(x,y)$ takes the two variables $x \in \mathcal{V}^{\mathcal{A}}$ and $y \in \mathcal{V}^{\mathcal{C}}$ as inputs and gets the penalty value between their sequences.

$$cand_4(x,y) := \text{penalty}(\mathit{seq}(\mathcal{T}^{\mathcal{A}},x), \mathit{seq}(\mathcal{T}^{\mathcal{C}},y))$$

Algorithm 1 shows the computation of the penalty value between the two given sequences s_1 and s_2. First, on line 2 the sequences are evaluated. That means each value maps to a sequence of indices. The indices give the position in the sequence where to find the mapped value. The indices determine the order of values in time.

Example 1. Given is the sequence $s = (1,2,3,2,1,3,1)$. The sequence is transformed to the mappings m:

$$m = [1 \mapsto (0,4,6), 2 \mapsto (1,3), 3 \mapsto (2,5)]$$

On line 3 variables are initialized which represent the penalty value p and the index variables i_1, i_2 for iteration over the sequences. Next, the penalty value of both sequences is computed in a loop until one sequence is completely considered. As first step in the loop on line 5, the currently observed values are loaded. In case both values are equal, a mapping is found and only the index variables i_1 and i_2 are incremented and the corresponding entries in m_1 and m_2 are removed. In case the values v_1 and v_2 differ, the penalty value is increased. The variables n_1 and n_2 shown on lines 11 and 12 offer the position when the value of the other sequence occurs next on the considered sequence. This is needed to determine the values of which sequence are skipped next. That means the indices n_1 and n_2 indicate where the next equal value occurs on the other trace, respectively. If a value does not occur in

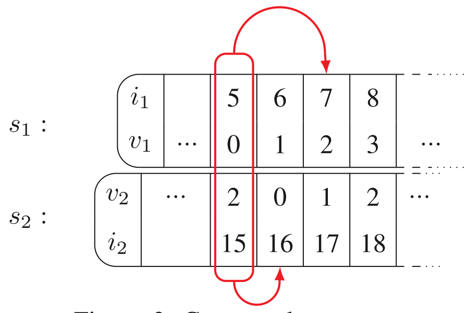


Figure 3: Compared sequences

other trace, then n_i is considered as infinity. The difference between the positions of the next occurrence of the considered value and the currently considered value $n_i - i_i$ determines the trace in which the value is skipped. After finding the value to be skipped, the corresponding indices are incremented on lines 14–18. The last step on line 20 adds the length of the unconsidered remainder of the sequences multiplied with a factor w to the penalty value. The factor is to give the remainder of the sequences a weight.

A minimal penalty value represents the best fit for the mapping between two considered variables, because a smaller penalty means that the sequences have a better match between each other.

Example 2. Fig. 3 illustrates a comparison step of Algorithm 1 if values are not equal. The inner rows (v_1, v_2) of the sequences show the values of the traces and the outer rows (i_1, i_2) show the indices of the currently compared values. In the red rectangle the currently considered values are displayed. For the first sequence s_1 the 5th and for the second sequence s_2 already the 15th value are compared. Next, it is analyzed when the currently considered values are occurring in the other trace, respectively, because the values v_1 and v_2 are different. The next time the value 2 is found in sequence s_1 is on position $n_1 = 7$ and the value 0 is found on position $n_2 = 16$ next time. Thus, one value in s_2 is skipped to get two equal values again. In the other case two values would be skipped. Consequently, i_2 is incremented and s_2 is considered at position 16, next.

Normalization of the fourth heuristic $cand_4(x, y)$ is done by $unify_4(x, y)$. The result of the division of $cand_4(x, y)$ and the worst existing value computed by the sequence heuristic is subtracted from 1. A value close to 1 represents a good fitting value, because less values are skipped during the comparison of x and y.

$$unify_4(x,y) := 1 - \frac{cand_4(x,y)}{worstSequenceValue}$$

III. Results

This section summarizes the experimental setup and shows the results. For experiments we applied our approach on two CPU designs. The first is the y86 processor[1]. A small processor for educational purposes. Next, we used the *OpenRISC 1000* (OR1k)[2]. For reduction of the runtime the best results of the intersection heuristic are used. It is assumed that the correct correspondences for the other methods will not falsify the results because two corresponding variables always have a large intersection of values. For the benchmarks a computer with an Intel i7-3520M CPU with 2.9 GHz and 8GB RAM is used.

[1]http://www.digitaltechnik.org/examples/Y86_seq.zip

Table I: Mapping of variables by heuristic for y86

	intersection	sym. difference	count	sequence
IP	IP (2)	IP (2)	IP (1)	IP (1)
I0	opcode (2)	opcode (2)	opcode (2)	opcode (2)
R[0]	eax (1)	eax (1)	eax (1)	eax (1)
R[3]	ebx (1)	ebx (1)	ebx (1)	ebx (1)
ZF	ZF (6)	ZF (4)	ZF (1)	ZF (1)

A. y86 Processor

This section deals with the experimental results for a y86 processor. This is a simple CPU design implementing a subset of the x86 32-bit instruction set architecture.

Table I shows the results of the matching for the y86 processor. The headline shows which heuristic is presented in the given column. The first column shows the starting variable for the mapping and the remaining columns show the results for the mapping. The number in parentheses behind the variable shows the number of candidates for the achieved best fit, i.e., largest value of *unify*().

The abstract *Instruction Pointer* (IP) variable is mapped to the corresponding IP variable in the concrete model. The intersection and symmetric difference approaches get also one further candidate for the IP but the count and sequence methods map to the corresponding IP variable only. The abstract model splits one instruction to three variables (I0, I1 and I2). The first part of the instruction is extracted to I0 and includes the opcode of the present instruction. All methods map the `opcode` to the opcode and `current_opcode` variable of the concrete model. Next, the variables R[0] and R[3] representing the first and the fourth register of the y86 processor. The reference shows [12] that these are the registers `eax` and `ebx`. Both registers are perfectly mapped by all methods. The other registers were not used in the given use cases. The last variable shown represents the *Zero Flag* (ZF) variable. The first two methods achieve results with more than one best fit candidate. That is because a flag variable is considered here that matches with all other flag variables or all other variables which get the values zero and one. Nevertheless, ZF is included in the set of candidates. The count and sequence methods are more precise and map to the correct ZF variable, only.

B. OpenRISC 1000

As a more complex example, the OR1k is used providing an ISS and RTL implementation. To execute both modules with the same stimuli a simple C/C++ program is implemented. These *use case*s are compiled to a binary file and are executed on both modules. To get the trace file of the concrete model $\mathcal{T}^\mathcal{C}$, a simulation of the CPU with the generated executable is performed. From the simulator a *Value Change Dump* (VCD) file is received which is considered as the trace of the concrete module. The abstract module for the OR1k is an ISS which is a very abstract model especially in comparison to other work [8][9] which consider less abstract models. For this reason no direct comparison to [8][9] can be carried out. To get the abstract trace file $\mathcal{T}^\mathcal{A}$, the ISS is instrumented by print statements after each assignment operation and variable initialization. For automatic generation, the *Abstract Syntax Tree* is used to include the print statements, which prints the name of the manipulated variable and its value. Next, the ISS is executed to get the trace file for the abstract model.

The use case is a C–program that implements a recursive function which computes the Fibonacci numbers. The function is called in a loop from 0 to 7. Table II shows an excerpt of

Table II: Mapping results for Fibonacci use case

		selected candidate			fit value	# of best fit cand.
		module	variable	set/seq size		
ISS ($\mathcal{A}$)		`execute.h`	`cpu_state.pc`	670/4298		
OR1k ($\mathcal{C}$)	symmetric diff.	`mor1kx_ctrl_cappuccino`	`pc_execute_i`	724	0.93	6
	intersection	`mor1kx_fetch_cappuccino`	*`pc_addr`*	745	1.00	23
	count	`mor1kx_ctrl_cappuccino`	`pc_execute_i`	724	0.73	11
	sequence	`mor1kx_ctrl_cappuccino`	`pc_execute_i`	6400	0.75	6

the results for the mapping from the abstract model (ISS) $\mathcal{A}$ to the concrete model (OR1k) $\mathcal{C}$. In total, data for 411 times 2372 variables for four heuristics is generated which need to be filtered.

Table II shows a selected number of mappings for the Fibonacci use case. The rows headed by ISS in the first column show the variable of the abstract model. The next four rows headed by OR1k show the mapped variables of the concrete model returned by each heuristic printed in the second column. The third column shows the module where the given variable comes from. Column four shows for the abstract model the starting variable for the mapping. The rows related to the concrete model show one selected candidate returned by the mapping heuristic. Next column (5th) shows the size of the related set or sequence for the given variable. The two last columns show the best fitting value of the selected variable and the number of variables that also have the best fitting value.

Table II shows the mapping for the PC variable which has a corresponding variable in both models. The intersection heuristic maps `cpu_state.pc` of the ISS to the register `pc_addr` in the Verilog implementation. This is the mapping already shown in Fig. 1 as an example. The `pc_addr`s module `mor1kx_fetch_cappuccino` contains the part where the PC is increased by one instruction. From the perspective of design understanding this is a very good result. Starting from the ISS in the `execute.c` file where the PC is increased a mapping to the module `mor1kx_fetch_cappuccino` is present. For the PC an intersection with 669 elements is achieved. Starting with a set of 670 values in the abstract model it is only a difference of one element. This means a fit value of 1. The concrete model has a set with 745 values. This difference can be explained by the abstraction of the ISS. In addition to `pc_addr` 23 more best fit elements are found but all are related to signals and registers of the PC.

The other heuristics map $6 - 10$ candidates, which are also related PC-variables in the pipeline. The heuristics have not found the `pc_addr` register as best fit but the next best fitting variables include this register also. By considering the next best fitting variables the number of candidates increases to 29. The results of the sequence heuristic show that the PC of the abstract model obtains 2102 values less than the concrete model.

The next selected mapping candidate is `cpu_state.insn_ea` which stores the effective address of the current instruction. The specification [13] says, that the effective address will be translated by the *Memory Management Unit* (MMU) to the physical address. So the effective address is a virtual address which needs to be translated. All heuristics map from `cpu_state.insn_ea` to the MMU of the OR1k and except for the count heuristic, all heuristics map to the output `virt_addr_match_i`. That means that the effective address of the abstract model maps to the virtual address input of the MMU in the concrete model. In case of the sequence heuristic both sequences have a matching of 100% which is an ideal matching outcome of the mappings.

The mapping for the registers obtains moderate results. To get better results, another use case is introduced. This use case simply assigns different values to each register and afterwards adds distinguishable values to the registers. This is done by writing explicitly to the registers using assembler code. This use case is especially introduced to get better results for the mapping of the registers. The results of the second use case show that directed use cases can improve the mapping results. For the PC similar results are achieved. The results for `cpu_state.insn_ea` are better than the Fibonacci use case. All heuristics map to the input of the MMU. All register variables (`reg[*]`) are mapping with the intersection heuristic to the input (`rfa/din`) and output signals of the register file module `mor1kx_rf_cappuccino`. The register file includes all registers in one memory block and the specific registers are addressed by additional inputs for the reading and writing address, respectively. This is an improvement of the mapping is achieved. All values of the register file module are the union of the values of all registers of the abstract model. In a design understanding point of view, a mapping from the registers to the register file module is ideal.

Fig. 4 shows average of the fit values for multiple use cases. Fig. 4 illustrates the results for three special registers (`sprs[*]`), a signal which stores the instruction (`cpu_insn`) and again the PC. For these results a third use case is introduced that calculates the factorial of the numbers from one to 14. Fig. 5 presents the number of calculated candidates with the corresponding fit value in Fig. 4.

For the mapping between `sprs[EEAR]` to `spr_eear` the figure shows the average fit value. The fit value for the first special register is always 0.33 and the number of matched candidates is 6 for the intersection and 2 for all other heuristics. Two candidates (of 2372 variables) is again a very good result. The next special register (`sprs[PPC]`) stores the *Previous Program Counter* (PPC) and is mapped to `spr_ppc`. The fit value are almost equal to the fit value of the PC (last in Fig. 4 and 5) but the best achieved number of candidates for the PPC special register is 4 while the PC's best result is 6 candidates. The fit values of the status register are presented with the middle bars. The fit value for the intersection heuristic is 1 with 36 more candidates. The other heuristics have a fit value of 0.57 with 436 candidates, 0.86 with 257 candidates and the sequence heuristic of 0.97 with 244 candidates together with the desired variable (`spr_sr`). The mapping for the instruction variable of the abstract model maps to a signal in a debug module which monitors the processor (`i_monitor/insn`). Here the heuristics compute exactly one candidate for the instruction variable.

As a next step, to improve the number of candidates, the fit values of the symmetric difference and the intersection heuristic are combined. For all mappings the average of the fit value is computed. Fig. 4 also shows the average fit value of the first two methods and Fig. 5 shows how the method also

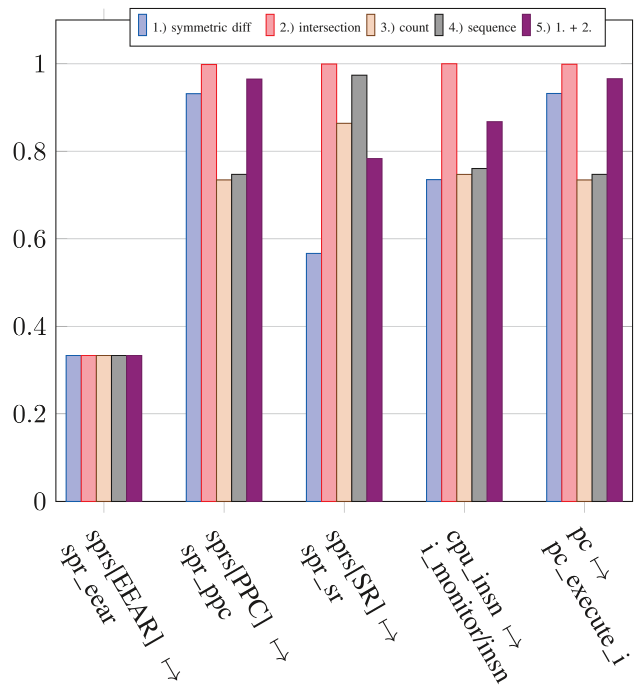


Figure 4: Mapping of selected variables with merged results

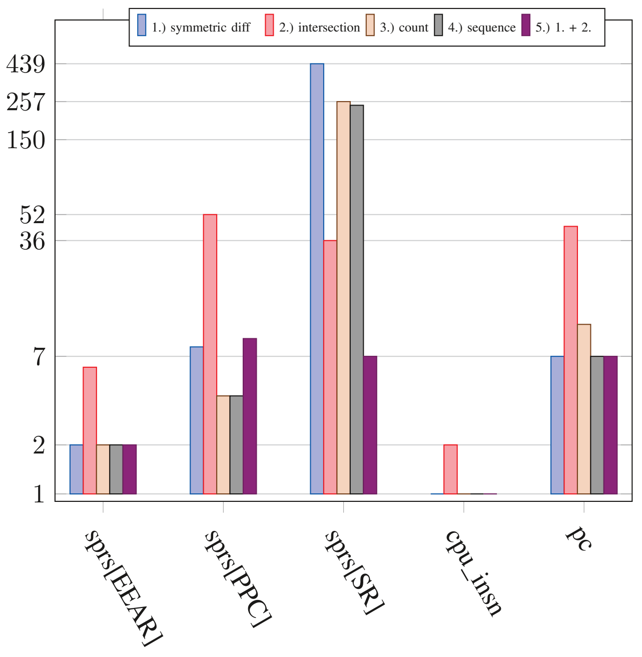


Figure 5: # of candidates for mappings with merged results

improves the number of candidates. For example the status register has in the worst case 439 candidates. By combining both heuristics for the status register only 7 candidates are obtained which reduces the number of candidates by 431.

Table III shows the average runtime in seconds for the mapping process that is currently not optimized. Each variable of the abstract model (411 variables) is compared with each variable of the concrete model (2372 variables). To calculate the average runtime, the experiment is repeated ten times. The first column shows the heuristics. The middle column presents the elapsed runtime for the mapping process with the C-program (Fibonacci) as the use case. In the last column the average time taken by the *register usage* program is shown.

The execution time for the register usage code is longer than for Fibonacci. The traces for the register usage program are longer and have more values, which need to be handled. That is the reason, why the mapping takes more time for the register usage code. However, the results show that directed use cases improve the results so that a longer runtime is worthwhile.

IV. Conclusion

A mapping of variables from a very abstract to a concrete model is possible despite the strong dissimilarities. This is confirmed by the presented results. The heuristics yield quite exact results even when comparing an ISS to an RTL implementation of a processor. This can provide an extremely useful tool for design understanding in a practical setting.

In future work other concepts will be considered, e. g., a machine learning approach which uses the results of multiple use cases. A further approach is to use approximate string matching when comparing signal traces.

Table III: Average runtime for mapping process (seconds)

heuristics	Fibonacci	register usage
symmetric diff.	8.0	56.4
intersection	16.0	38.3
count	9.9	58.7
sequence	28.0	108.4

References

[1] K. Hao, F. Xie, S. Ray, and J. Yang, “Optimizing equivalence checking for behavioral synthesis,” in *Design, Automation and Test in Europe*, 2010, pp. 1500–1505.

[2] S. Vasudevan, V. Viswanath, J. A. Abraham, and J. Tu, “Sequential equivalence checking between system level and RTL descriptions,” *Design Automation for Embedded Systems*, vol. 12, pp. 377–396, 2008.

[3] A. Finder, J. Witte, and G. Fey, “Debugging HDL designs based on functional equivalences with high-level specifications,” in *IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2013, pp. 60–65.

[4] B. Alizadeh and M. Fujita, “Automatic merge-point detection for sequential equivalence checking of system-level and RTL descriptions,” in *Proceedings of the International Conference on Automated Technology for Verification and Analysis*, 2007, pp. 129–144.

[5] X. Feng and A. J. Hu, “Early cutpoint insertion for high-level software vs. RTL formal combinational equivalence verification,” in *Proceedings of Design Automation Conference*, 2006, pp. 1063–1068.

[6] C. Karfa, C. Mandal, D. Sarkar, S. R. Pentakota, and C. Reade, “A formal verification method of scheduling in high-level synthesis,” in *Proceedings of the International Symposium on Quality Electronic Design*, 2006, pp. 71–78.

[7] S. Vasudevan, J. Abraham, V. Viswanath, and J. Tu, “Automatic decomposition for sequential equivalence checking of system level and RTL descriptions,” in *Formal Methods and Models for Co-Design*, 2006, pp. 71–80.

[8] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur, and N. Sharma, “Non-cycle-accurate sequential equivalence checking,” in *Design Automation Conference*, 2009, pp. 460–465.

[9] D. Große, M. Groß, U. Kühne, and R. Drechsler, “Simulation-based equivalence checking between SystemC models at different levels of abstraction,” in *Great Lakes Symposium on VLSI*, 2011, pp. 223–228.

[10] IEEE Standards Association, “IEEE Standard for Standard SystemC Language Reference Manual,” *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, Jan 2012.

[11] M. Fujita, Y. Kojima, and A. M. Gharehbaghi, “Debugging from high level down to gate level,” in *Design Automation Conference*, 2009, pp. 627–630.

[12] A. Biere, D. Kröning, G. Weissenbacher, and C. Wintersteiger, *Digitaltechnik - Eine praxisnahe Einführung*, ser. Springer-Lehrbuch. Springer Berlin Heidelberg, 2008.

[13] OpenCores Community, “OpenRISC 1000 Architecture Manual,” 2014.