Real-Time Systems, 32, 253–277, 2006
© 2006 Springer Science + Business Media, Inc. Manufactured in The Netherlands.
DOI: 10.1007/s11241-005-4683-4

Code Analysis for Temporal Predictability*

JAN GUSTAFSSON jan.gustafsson@mdh.se
BJÖRN LISPER bjorn.lisper@mdh.se
Department of Computer Science and Electronics, Mälardalen University, Västerås, Sweden

RAIMUND KIRNER raimund@vmars.tuwien.ac.at
PETER PUSCHNER peter@vmars.tuwien.ac.at
Institut für Technische Informatik, Technische Universität Wien, Austria

Published online: 3 March 2006

Abstract. The execution time of software for hard real-time systems must be predictable. Further, safe and not overly pessimistic bounds for the worst-case execution time (WCET) must be computable. We conceived a programming strategy called WCET-oriented programming and a code transformation strategy, the single-path conversion, that aid programmers in producing code that meets these requirements. These strategies avoid and eliminate input-data dependencies in the code. The paper describes the formal analysis, based on abstract interpretation, that identifies input-data dependencies in the code and thus forms the basis for the strategies provided for hard real-time code development.

Keywords: worst-case execution time analysis, real-time languages, compiler optimizations, code transformation, abstract interpretation, graph transformation

1. Introduction

One of the central demands on a real-time system is that it meets all timing requirements imposed by the application under guarantee. In order to give such a guarantee about the temporal correctness, engineers use specific design, implementation, and verification techniques. These techniques have to ensure the correct timing of entire applications down to the level of single tasks.

At the single-task level, a development framework has to aid the real-time programmer in a number of ways. Most essentially, the programmer needs to be provided with strategies for developing code with timing that is both predictable and easy to analyze. Further, techniques and tools for assessing the timing (e.g., WCET) of the code are needed. These tools need to provide the programmer with high-quality feedback about the temporal properties of the code.

We worked out a strategy for real-time task development that includes the above-mentioned services to the programmer: Tool support for *WCET-oriented programming* (Puschner, 2003) advises the programmer to write code that avoids algorithms or coding of producing code with a heavily input-data dependent control flow. Further, the *single-path conversion* (Puschner and Burns, 2002; Puschner, 2002) converts the remaining or unavoidable input-data dependent alternatives in the control flow into code with a

*This work has been supported by the ARTIST2 Network of Excellence on Embedded Systems Design of IST FP6.

unique execution trace. The remaining code has a fully predictable timing and is easy to analyze for its WCET.

This paper describes techniques that can be used to implement tool support for WCET-oriented programming and the single-path conversion. Both WCET-oriented programming and single-path conversion rely on an effective analysis of input-data dependencies in control decisions in the code. We show how this analysis is realized by abstract interpretation and present the formal framework of the analysis in detail.

The paper is structured as follows: Section 2 gives a summary on WCET-oriented programming and single-path conversion. Section 3 introduces abstract interpretation and Section 4 describes the **While** language, a simple programming language that we use to define the semantics of our interpretation. Section 5 defines the abstract domain, states, and rules of the interpretation used for the input-dependency analysis. Sections 6 and 7 illustrate our approach with examples. A correctness proof of our analysis is given in Section 9 and Section 10 concludes the paper.

2. WCET-Oriented Programming and Single-Path Conversion

The requirements imposed on real-time code for safety-critical systems are very different from what is expected from non real-time code. While functional correctness is required in both areas, non real-time code is usually expected to be laid out for good average performance, i.e., typical executions must return results quickly; longer execution times are acceptable if they occur rarely. For Real-time code, in contrast, average performance is not of primary concern. Real-time code (for safety-critical applications) is expected to have a short guaranteed worst-case execution time and small execution-time jitter. Further, we expect from real-time code that is has predictable control flow allowing for an unquestionable assessment of how the program will perform under the specified conditions.

We argue that in order to achieve these very different properties of hard real-time code—a short WCET, small execution-time jitter, and traceability of timing—the algorithms and the coding used in hard real-time systems need to be very different from those algorithms and structures being traditionally used. We therefore propose two mechanisms: *WCET-oriented programming*, a new way to selecting and writing algorithms and code for real-time systems (Puschner, 2003), and the *single-path code conversion*, a code transformation that yields jitter-free or low-jitter code that is easy to analyze for its WCET (Puschner and Burns, 2002).

WCET-Oriented Programming. The goal of WCET-Oriented Programming is to produce code with a very simple control structure that avoids input-data dependent control-flow decisions as far as possible. When we look at traditional code that has been laid out for good performance, we will find that avoiding input-data dependent control flow is rather atypical. On the contrary, in well performing algorithms the use of input-data dependent control decisions is the central key to achieving high speed for the most frequent scenarios, thus strongly determining the overall performance of the algorithm. Often, such performance-oriented algorithms perform very poorly for the unprobable and rare scenarios. This is, however, not a problem in a performance-oriented world.

In hard real-time systems, the situation is very different. The primary "performance" measure for hard real-time code is its worst-case execution-time. This different measure of performance necessarily asks for a different way of coding solutions. In *WCET-oriented programming*, it is no longer meaningful to favor certain execution scenarios of an application over the others. What counts is that the overall worst-case execution time is minimized. WCET-oriented programming achieves this by trying to using or producing algorithms being free from input-data dependent control flow decisions or, if this cannot be completely achieved, by restricting the number of operations being only executed for a subset of the input-data space to a minimum.

WCET-oriented programming yields algorithms and program code that look quite unconventional, as programmers are so much used to solutions being optimized for the general case. It is therefore necessary to provide tool support for WCET-oriented programming. Such programming support can be offered, for example, in the form of intelligent editors that help programmers to identify input-data dependent control flow (Fauster et al., 2003) or in the form of tools that transform input-data-dependent control flow patterns into code with a static control flow not depending on inputs.

Besides keeping the WCET low, WCET-oriented code has the following advantages: Due to the reduction, or - if possible - elimination of data-dependent control flow, the traces of all executions are (almost) identical. As a consequence, the execution times of all executions are similar and the execution-time jitter is small. Further, the absence (or reduction) of input-data dependent control flow makes keeps the total number of different execution paths through the code low. For such code it is easier to argue about possible behaviors and execution times of program code than for very complex code whose behavior is very sensitive to the actual input values. In this way, WCET-oriented programming does not only produce code with better WCETs but also yields better traceable and more dependable WCET-analysis results than traditional programming.

The Single-Path Conversion. The single-path conversion is a code translation that transforms code into new code with constant execution time. The single-path conversion has been conceived with similar goals in mind as WCET-oriented programming – producing jitter-free code that is easy to analyze for its WCET. Still, the single-path conversion and WCET-oriented programming are two different and independent concepts, even though they might be used together: WCET-oriented programming describes an approach towards designing algorithms and writing software. The single-path conversion, in contrast, is applied after code implementation. It takes an arbitrary piece of real-time code, no matter whether it is WCET-oriented or not, and transforms it into code with constant timing.

The strategy of the single-path conversion is to obtain a constant code execution time by rearranging the code in a way that it ends up with a single execution trace. Therefore, the process is named as the single-path conversion. Similar to the WCET-oriented approach, the idea of the single-path conversion is to remove input-data dependencies in the control flow. The single-path conversion, however, does so by replacing all input-data dependent branching operations in the code by predicated code, i.e., it puts the input dependent alternatives in sequence into a sequential pieces of code and uses predicates (instead of branches) and, if necessary, speculative execution to select the right code to be executed at runtime (Mahlke et al., 1995). For pieces of code with an if-then-else semantics, similar transformations have been used before to avoid pipeline stalls in processors with deep pipelines. This technique is called if-conversion (Allen et al., 1983). In addition to code with if-then-else semantics the single-path conversion also transforms loops with input-data dependent control conditions. This transformation that yields loops with constant iteration counts has been described in Puschner (2002).

The single-path conversion is very generally applicable to hard real-time code. A prerequisite that needs to be fulfilled in order to transform a piece of code, however, is that upper bounds for the number of iterations of all loops have to be provided to the conversion in some way. This can either be done by semantic analysis of the code or in the form of annotations given by the user, in case an automated analysis is not possible or available. Further, if single-path code is expected to execute with invariable execution time then it needs to be supported by appropriate hardware, i.e., hardware whose instruction timing does neither depend on operand nor on predicate values (see Puschner and Burns, 2002).

As we mentioned above, WCET-oriented programming and the single-path conversion are different concepts and, in principle, independent. On the other hand, the two concepts do nicely fit together to produce code that is both free of input-data dependent control decisions and well performing. WCET-oriented programming supports the programming strategy to write code with good worst-case performance, a clear structure, and stable execution times. In some cases, however, the semantics of a given problem or the limitations of the programming language used will not allow the programmer to write code that treats all input scenarios identically. In these cases, the single-path conversion provides the mechanism to remove the remaining input-data dependent control flows from the code and achieve fully constant timing.

3. Abstract Interpretation

Our aim is to calculate the run-time behavior of a program without having to run it on *all* input data, and while guaranteeing termination of the analysis.

One such technique for program analysis is *abstract interpretation* (Cousot and Cousot, 1977; Gustafsson, 2000), which means to calculate the program behavior using value descriptions or *abstract values* instead of real values. The price to be paid is loss of information; the calculation will sometimes give only approximate information. Abstract interpretation has three important properties:

1. It yields an *approximate* and *safe* description of the program behavior.
2. It is *automatic*, i.e., the program does not have to be annotated.
3. It works for *all* programs in the selected language.

It is important that the approximations for the concrete values are selected to *reduce the necessary calculations* in each step. But, in general, loss of precision is often the consequence of less calculation effort.

4. The While Language

For the sake of simplicity, we use the **While** language (Nielson and Nielson, 1992), with a syntax and semantics similar to common imperative languages like C and Pascal. We have added arrays to the language, to be able to handle the larger example in Section 7.

Syntax. The **While** language is built from the following syntactic sets:

- the set of statements, STATEMENT.
- the set of arithmetic expressions, AEXP;
- the set of boolean expressions, BEXP;
- the set of variables in the program, VARIABLE = {a, ... , z};
- the set of numerals, NUM; and
- the truth values, $T = \{tt, ff\}$;

We use the following meta-variables when we describe the semantics of **While**:

- $var \in$ VARIABLE;
- $n \in$ NUM;
- $a \in$ AEXP;
- $b \in$ BEXP; and
- $S \in$ STATEMENT.

We use the following syntactical formation rules for programs written in **While** (where typewriter text represents source text, and italics represents meta-variables):

$$\begin{aligned}
\text{AEXP} &::= n \mid var \mid var[a] \mid a_1{+}a_2 \mid a_1 - a_2 \\
\text{BEXP} &::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1\ <\ a_2 \mid \\
&\quad\ a_1\ <=\ a_2 \mid a_1\ >\ a_2 \mid a_1\ >=\ a_2 \mid \\
&\quad\ \texttt{not}(b) \mid b_1 \& b_2 \\
\text{STATEMENT} &::= var := a \mid var[a_1] := a_2 \mid \\
&\quad\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \\
&\quad\ \texttt{while}\ b\ \texttt{do}\ S \mid S_1; S_2 \mid \texttt{skip}
\end{aligned}$$

State. The state σ is a mapping from variables to values (integers). Each element in an array is regarded as a separate variable. We use the notation $\sigma = [\mathtt{x} \mapsto 0]$ to denote a state where x is assigned the value 0. Since σ is a mapping from variables to values, the expression $\sigma(\mathtt{x})$ will give the result 0. With $\sigma_1 = \sigma[\mathtt{x} \mapsto 1]$ we denote the updated state σ_1 where x is re-assigned to the value 1.

Semantics. We will use a denotational semantics for **While**, as shown in Figures 1 and 2. For further details on this semantics, we refer to Nielson and Nielson (1992). We have added rules for arrays, to be able to handle the example in Section 7.

In the semantics,

- $\mathcal{N}$ defines the meaning of numerals $n \in$ NUM;
- $\mathcal{A}$ the meaning of arithmetic expressions $a \in$ AEXP;

$$\begin{array}{rcl} \mathcal{A}[\![n]\!]\sigma & = & \mathcal{N}[\![n]\!] \\ \mathcal{A}[\![var]\!]\sigma & = & \sigma(var) \\ \mathcal{A}[\![var[a]]\!]\sigma & = & \sigma(var[\mathcal{A}[\![a]\!]\sigma]) \\ \mathcal{A}[\![a_1{+}a_2]\!]\sigma & = & \mathcal{A}[\![a_1]\!]\sigma + \mathcal{A}[\![a_2]\!]\sigma \\ \mathcal{A}[\![a_1{-}a_2]\!]\sigma & = & \mathcal{A}[\![a_1]\!]\sigma - \mathcal{A}[\![a_2]\!]\sigma \end{array}$$

$$\begin{array}{rcl} \mathcal{B}[\![\mathtt{true}]\!]\sigma = tt & & \mathcal{B}[\![\mathtt{false}]\!]\sigma = \mathit{ff} \\ \mathcal{B}[\![a_1 = a_2]\!]\sigma & = & \mathcal{A}[\![a_1]\!]\sigma = \mathcal{A}[\![a_2]\!]\sigma \\ \mathcal{B}[\![a_1 < a_2]\!]\sigma & = & \mathcal{A}[\![a_1]\!]\sigma < \mathcal{A}[\![a_2]\!]\sigma \\ \mathcal{B}[\![a_1 <= a_2]\!]\sigma & = & \mathcal{A}[\![a_1]\!]\sigma \leq \mathcal{A}[\![a_2]\!]\sigma \\ \mathcal{B}[\![a_1 > a_2]\!]\sigma & = & \mathcal{A}[\![a_1]\!]\sigma > \mathcal{A}[\![a_2]\!]\sigma \\ \mathcal{B}[\![a_1 >= a_2]\!]\sigma & = & \mathcal{A}[\![a_1]\!]\sigma \geq \mathcal{A}[\![a_2]\!]\sigma \\ \mathcal{B}[\![\mathtt{not}(b)]\!]\sigma & = & \neg\mathcal{B}[\![b]\!]\sigma \\ \mathcal{B}[\![b_1 \& b_2]\!]\sigma & = & \mathcal{B}[\![b_1]\!]\sigma \wedge \mathcal{B}[\![b_2]\!]\sigma \end{array}$$

Figure 1. Semantics of expressions.

$$\begin{array}{rcl} \mathcal{S}[\![\mathtt{x} := a]\!] & = & \lambda\sigma.\sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \\ \mathcal{S}[\![var[a_1] := a_2]\!] & = & \lambda\sigma.\sigma[var[\mathcal{A}[\![a_1]\!]\sigma] \mapsto \mathcal{A}[\![a_2]\!]\sigma] \\ \mathcal{S}[\![\mathtt{skip}]\!] & = & \lambda\sigma.\sigma \\ \mathcal{S}[\![\mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2]\!] & = & \lambda\sigma.\mathbf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathbf{then}\ \mathcal{S}[\![S_1]\!]\sigma\ \mathbf{else}\ \mathcal{S}[\![S_2]\!]\sigma \\ \mathcal{S}[\![\mathtt{while}\ b\ \mathtt{do}\ S]\!] & = & \mathit{fix}\ (\lambda g.\lambda\sigma.\mathbf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathbf{then}\ g(\mathcal{S}[\![S]\!]\sigma)\ \mathbf{else}\ \sigma) \\ \mathcal{S}[\![S_1; S_2]\!] & = & \mathcal{S}[\![S_2]\!] \circ \mathcal{S}[\![S_1]\!] \end{array}$$

Figure 2. Semantics of statements.

- $\mathcal{B}$ the meaning of boolean expressions $b \in$ BEXP; and
- $\mathcal{S}$ the meaning of statements $S \in$ STATEMENT.

Control Flow Graph and Data Flow Functions. We can use the semantic functions in **While** to define functions from states to states in the control flow. These *data flow functions* can be visualized in control graph structures (see Figure 3).

5. Abstract Interpretation of While Programs to find Input Dependencies

The purpose of the program analysis is to find the properties of conditions in programs. A condition is defined to depend on input data if any of the variables included in the condition is input dependent. This means that the analysis first calculates input data dependencies for variables, then draws conclusions about the conditions. We differ between two cases:

1. The condition (e.g., in if, switch and loop-statements) is *not* depending on input data.

2. The condition *may* depend on input data.

For the first case, the code generation may generate ordinary code, while for the second case, single path conversion using predicated instructions (Puschner, 2002) has to be used to assure single path behavior.

Data Flow Function

Assign and skip: $\sigma_2 = [\![S]\!]\,\sigma_1$

$\sigma_2 = [\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\,\sigma_1$

$\sigma_2 = [\![\texttt{while}\ b\ \texttt{do}\ S]\!]\,\sigma_1$

$\sigma_2 = [\![S_1;\ S_2]\!]\,\sigma_1$

Control Flow Graph

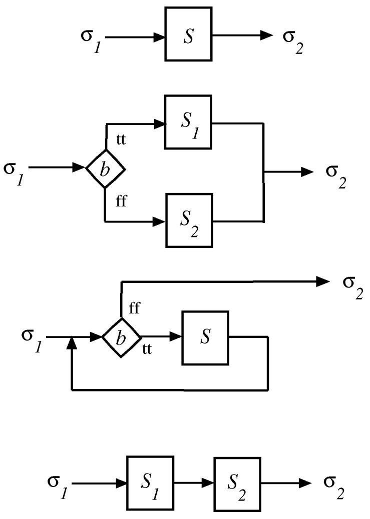


Figure 3. Data flow functions and control flow graphs.

5.1. Abstract Program

The analysis is a *semantics-based abstract interpretation*. We will, as a preparation to the abstract interpretation, transform a program to a corresponding abstract program. Interpretation of this abstract program, using abstract semantic rules and abstract values, will yield a final value for the variables in each program point, which is a safe approximation of the corresponding concrete values. “Safe” in this context means that all real executions are always “covered” by the abstract executions.

We can use this information to, in a safe way, determine whether ordinary or predicated instructions should be generated for the conditions in the program. For the two cases mentioned above, safety means that ordinary code is always correctly generated for input data dependent conditions. Sometimes, however, predicated instructions may be generated for non-input dependent conditions due to overestimations by the abstract interpretation.

5.2. Abstract Domain

All abstract variables in the program will be mapped to the abstract domain below. This mapping is called the *abstract state* and is denoted with $\tilde{\sigma}$.

$$\begin{tikzcd}\texttt{ID} \arrow[d, dash] \\ \texttt{NID}\end{tikzcd}$$

The values in the Hasse diagram have the following explanation:

- *ID* marks a value that may be input dependent;
- *NID* marks a value that is not input dependent.

The order in the diagram represents the information content of the value. Also, the order defines the effect of the least upper bound ($\sqcup_{var}$) operation, used for disjunctions of variable values. For example, if a value is *ID* or *NID*, the result will be the safe (over)approximation $ID \sqcup_{var} NID = ID$.

The least upper bound for abstract states ($\sqcup$) is defined as $\sqcup_{var}$ for each variable. Formally

$$(\tilde{\sigma}_1 \sqcup \tilde{\sigma}_2)(var) = \tilde{\sigma}_1(var) \sqcup_{var} \tilde{\sigma}_2(var)$$

for all variables *var* in the resulting state.

5.3. Initial Abstract State of a Program

In the initial abstract state $\tilde{\sigma}_0$ of a program, the abstraction function α will set all input data dependent variables (corresponding to e.g., input parameters to C functions) to *ID*, and the rest of the variables to *NID*.

Arrays are abstracted in a special way. For each concrete array (e.g. `a[`i`]`, $i = 1, \ldots, n$) there is *one* abstract variable `a[]` representing *all* values in `a`. The reason for this is simplicity—the simple abstract domain means that we cannot keep track of the indices in arrays. At initialization, for each of its elements `a[i]` we first calculate the abstract value $\alpha($`a[i]`$)$ as for ordinary variables. Then we set `a[]` $= \sqcup_{i=1}^{n}(\alpha($`a[`$i$`]`$))$.

We also add an extra variable to the state, `flow`, representing the data dependency of the control flow in the program at the current program point. We will set `flow` to *NID* at the beginning of the program. Actually, it will stay set to *NID* until the control flow is somehow controlled by an input data dependent variable.

5.4. Abstract Semantic Functions

For a given program, a system of abstract data flow equations can be set up based on the control graph of the program and the abstract semantic functions, as defined below. In the equations, $\tilde{\mathcal{S}}[\![S]\!]$ denotes the abstract semantic function of S. The initial abstract state is assigned to the input edge of the program.

Primitive Abstract Semantic Functions. First we define the abstract transition functions for the primitive (i.e., not compound) statements, corresponding to the first line of Figure 3.

Abstract assignment to variable ($var := a$) is similar to concrete assign:

$$\tilde{\mathcal{S}}[\![var := a]\!]\tilde{\sigma} = \tilde{\sigma}[var \mapsto \tilde{\mathcal{A}}[\![a]\!]\tilde{\sigma}]$$

Abstract assignment to array ($var[a_1] := a_2$) assigns a value to `a[]` which is a safe overestimation of all possible values in the array. We also have to include the case when the index is input dependent:

$$\tilde{\mathcal{S}}[\![var[a_1] := a_2]\!]\tilde{\sigma} = \tilde{\sigma}[var[] \mapsto (\tilde{\mathcal{A}}[\![a_1]\!]\tilde{\sigma} \sqcup \tilde{\mathcal{A}}[\![a_2]\!]\tilde{\sigma} \sqcup \tilde{\sigma}(var[]))]$$

Skip is simple as always:

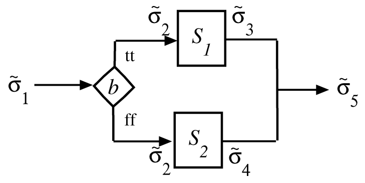
$$\tilde{\mathcal{S}}[\![\texttt{skip}]\!]\tilde{\sigma} = \tilde{\sigma}$$

Compound Abstract Semantic Functions. The compound functions are built up from the primitive functions. For the if and while statements, we need to define what happens in the merge point.

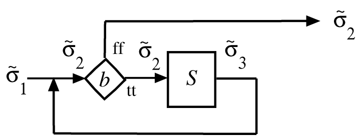
If the value of the condition `b` may be input dependent (*ID*), we set `flow` to *ID* for both edges. Then we analyze both possible edges and form the least upper bound of the results. When the analysis of the expression is finished, the value of `flow` is reset to its original value. The reason for this handling of `flow` is illustrated by the example in Section 6.

If the value of the condition `b` is *NID*, we simply analyze both possible edges and form the least upper bound of the results.

$\tilde{\mathcal{S}}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\tilde{\sigma}_1 = \tilde{\sigma}_5$ is defined by:



$$\tilde{\sigma}_5 = \begin{cases} (\tilde{\sigma}_3 \sqcup \tilde{\sigma}_5)[\texttt{flow} \mapsto \tilde{\sigma}_1(\texttt{flow})] & \text{if } \tilde{B}[\![b]\!]\tilde{\sigma}_1 = ID \\ \quad \text{where} & \\ \quad \tilde{\sigma}_2 = \tilde{\sigma}_1[\texttt{flow} \mapsto ID], & \\ \quad \tilde{\sigma}_3 = \tilde{[\![} S_1 \tilde{]\!]}\tilde{\sigma}_2 \text{ and} & \\ \quad \tilde{\sigma}_4 = \tilde{[\![} S_2 \tilde{]\!]}\tilde{\sigma}_2 & \\ \tilde{\sigma}_3 \sqcup \tilde{\sigma}_4 & \text{if } \tilde{B}[\![b]\!]\tilde{\sigma}_1 = NID \\ \quad \text{where} & \\ \quad \tilde{\sigma}_2 = \tilde{\sigma}_1, & \\ \quad \tilde{\sigma}_3 = \tilde{[\![} S_1 \tilde{]\!]}\tilde{\sigma}_2 \text{ and} & \\ \quad \tilde{\sigma}_4 = \tilde{[\![} S_2 \tilde{]\!]}\tilde{\sigma}_2 & \end{cases}$$



$\tilde{\mathcal{S}}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]\tilde{\sigma}_1 = \tilde{\sigma}_2$ is defined by:

$$\tilde{\sigma}_2 = \begin{cases} (\tilde{\sigma}_1 \sqcup \tilde{\sigma}_4)[\texttt{flow} \mapsto \tilde{\sigma}_2(\texttt{flow})] & \text{if } \tilde{B}[\![b]\!]\tilde{\sigma}_2 = \mathit{ID} \\ \quad \text{where} & \\ \quad \tilde{\sigma}_3 = \tilde{\sigma}_2[\texttt{flow} \mapsto \mathit{ID}] \text{ and} & \\ \quad \tilde{\sigma}_4 = [\![\tilde{S}]\!]\tilde{\sigma}_3 & \\ \tilde{\sigma}_1 \sqcup \tilde{\sigma}_4 & \text{if } \tilde{B}[\![b]\!]\tilde{\sigma}_2 = \mathit{NID} \\ \quad \text{where} & \\ \quad \tilde{\sigma}_3 = \tilde{\sigma}_2 \text{ and} & \\ \quad \tilde{\sigma}_4 = [\![\tilde{S}]\!]\tilde{\sigma}_3 & \end{cases}$$

$\tilde{\mathcal{S}}[\![S_1; S_2]\!]$ is defined by:

$$\tilde{\mathcal{S}}[\![S_1; S_2]\!]\tilde{\sigma} = \tilde{\mathcal{S}}[\![S_2]\!](\tilde{\mathcal{S}}[\![S_1]\!]\tilde{\sigma})$$

Solution of the Equations. We can solve the system of data flow equations by using Jacobi iteration, i.e., by iteration until a fixpoint is reached. Given a set of data flow equations f_i so that $\tilde{\sigma}_i = f_i(\tilde{\sigma}_1, \dots, \tilde{\sigma}_n)$ for $i = 2, 3 \dots n$ and defining F as

$$\begin{aligned} F(\tilde{\sigma}_1, \dots, \tilde{\sigma}_n) = (y_1, \dots, y_n) \text{ where } y_1 &= \tilde{\sigma}_0 \text{ and} \\ y_i &= f_i(\tilde{\sigma}_1, \dots, \tilde{\sigma}_n) \text{ for } i = 2, 3 \dots n \end{aligned}$$

we can obtain the least fixed point of F as the least upper bound of the ascending chain

$$\perp_G, F(\perp_G), F(F(\perp_G)), \dots F^n(\perp_G)$$

where

$$\perp = \lambda \mathit{var}.\mathit{var} \mapsto \mathit{NID}$$

and $\perp_G$ is $(\perp, \dots, \perp)$. That is, the calculation is started by setting all variables in all abstract states to the bottom element *NID* in the domain. The chain is finite since the domain used is finite. Therefore, we are guaranteed to find a solution with a finite number of steps.

6. Introductory Example

A simple example will show the use of the analysis, especially the handling of the `flow` variable. Let's analyze the program S below:

```
if x = 1 then z := 1 else z := 2;
if y = 1 then x := 1 else x := 2
```

with the initial state

$$\tilde{\sigma}_0 = [\texttt{x} \mapsto \mathit{ID}, \texttt{y} \mapsto \mathit{NID}, \texttt{z} \mapsto \mathit{NID}, \texttt{flow} \mapsto \mathit{NID}].$$

For this example, using the abstract semantic functions, we get the flow equations as shown in Figure 6. We have simplified some of the expressions by using rules from Figure 4. The abstract states in the equations refer to the corresponding states in Figure 5, which shows the control flow graph of the program.

$$\tilde{A}[\![n]\!]\tilde{\sigma} = \begin{cases} \mathit{NID} & \text{if } \tilde{\sigma}(\texttt{flow}) = \mathit{NID} \\ \mathit{ID} & \text{otherwise} \end{cases}$$

$$\tilde{A}[\![\mathit{var}]\!]\tilde{\sigma} = \begin{cases} \tilde{\sigma}(\mathit{var}) & \text{if } \tilde{\sigma}(\texttt{flow}) = \mathit{NID} \\ \mathit{ID} & \text{otherwise} \end{cases}$$

$$\tilde{A}[\![\mathit{var}[a]]\!]\tilde{\sigma} = \begin{cases} \tilde{\sigma}(\mathit{var}[\,]) & \text{if } \tilde{\sigma}(\texttt{flow}) = \mathit{NID} \\ & \text{and } \tilde{A}[\![a]\!]\tilde{\sigma} = \mathit{NID} \\ \mathit{ID} & \text{otherwise} \end{cases}$$

$$\tilde{A}[\![a_1 + a_2]\!]\tilde{\sigma} = \tilde{A}[\![a_1 - a_2]\!]\tilde{\sigma} = \tilde{A}[\![a_1]\!]\tilde{\sigma} \sqcup \tilde{A}[\![a_2]\!]\tilde{\sigma}$$

$$\tilde{B}[\![\texttt{true}]\!]\tilde{\sigma} = \tilde{B}[\![\texttt{false}]\!]\tilde{\sigma} = \begin{cases} \mathit{NID} & \text{if } \tilde{\sigma}(\texttt{flow}) = \mathit{NID} \\ \mathit{ID} & \text{otherwise} \end{cases}$$

$$\begin{aligned} &\tilde{B}[\![a_1 = a_2]\!]\tilde{\sigma} = \tilde{B}[\![a_1 < a_2]\!]\tilde{\sigma} = \tilde{B}[\![a_1 \le a_2]\!]\tilde{\sigma} = \\ &\tilde{B}[\![a_1 > a_2]\!]\tilde{\sigma} = \tilde{B}[\![a_1 \ge a_2]\!]\tilde{\sigma} = \\ &\qquad \tilde{A}[\![a_1]\!]\tilde{\sigma} \sqcup \tilde{A}[\![a_2]\!]\tilde{\sigma} \end{aligned}$$

$$\tilde{B}[\![\neg b]\!]\tilde{\sigma} = \tilde{B}[\![b]\!]\tilde{\sigma}$$

$$\tilde{B}[\![b_1 \wedge b_2]\!]\tilde{\sigma} = \tilde{B}[\![b_1]\!]\tilde{\sigma} \sqcup \tilde{B}[\![b_2]\!]\tilde{\sigma}$$

Figure 4. Abstract semantics of expressions.

$S = S_1; S_2$

$S_1 =$ `if` b_1 `then` S_3 `else` S_4

$b_1 =$ `x = 1`

$S_3 =$ `z := 1`

$S_4 =$ `z := 2`

$S_2 =$ `if` b_2 `then` S_5 `else` S_6

$b_2 =$ `y = 1`

$S_5 =$ `x := 1`

$S_6 =$ `x := 2`

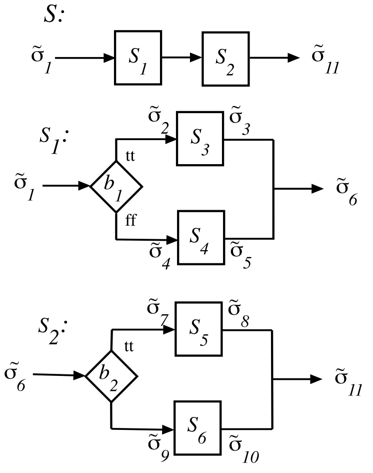


Figure 5. Control flow graphs for the example.

$$\tilde{\sigma}_1 = \tilde{\sigma}_0$$

$$\tilde{\sigma}_2 = \begin{cases} \tilde{\sigma}_1 \text{ if } \tilde{B}[\![\texttt{x} = \texttt{1}]\!]\tilde{\sigma}_1 = \mathit{NID} \\ \tilde{\sigma}_1[\texttt{flow} \mapsto \mathit{ID}] \text{ if } \tilde{B}[\![\texttt{x} = \texttt{1}]\!]\tilde{\sigma}_1 = \mathit{ID} \end{cases} \qquad \tilde{\sigma}_7 = \begin{cases} \tilde{\sigma}_6 \text{ if } \tilde{B}[\![\texttt{y} = \texttt{1}]\!]\tilde{\sigma}_6 = \mathit{NID} \\ \tilde{\sigma}_6[\texttt{flow} \mapsto \mathit{ID}] \text{ if } \tilde{B}[\![\texttt{y} = \texttt{1}]\!]\tilde{\sigma}_6 = \mathit{ID} \end{cases}$$

$$\tilde{\sigma}_3 = \tilde{\sigma}_2[\texttt{z} \mapsto \tilde{A}[\![1]\!]\tilde{\sigma}_2] \qquad \tilde{\sigma}_8 = \tilde{\sigma}_7[\texttt{x} \mapsto \tilde{A}[\![1]\!]\tilde{\sigma}_7]$$

$$\tilde{\sigma}_4 = \begin{cases} \tilde{\sigma}_1 \text{ if } \tilde{B}[\![\texttt{x} = \texttt{1}]\!]\tilde{\sigma}_1 = \mathit{NID} \\ \tilde{\sigma}_1[\texttt{flow} \mapsto \mathit{ID}] \text{ if } \tilde{B}[\![\texttt{x} = \texttt{1}]\!]\tilde{\sigma}_1 = \mathit{ID} \end{cases} \qquad \tilde{\sigma}_9 = \begin{cases} \tilde{\sigma}_6 \text{ if } \tilde{B}[\![\texttt{y} = \texttt{1}]\!]\tilde{\sigma}_6 = \mathit{NID} \\ \tilde{\sigma}_6[\texttt{flow} \mapsto \mathit{ID}] \text{ if } \tilde{B}[\![\texttt{y} = \texttt{1}]\!]\tilde{\sigma}_6 = \mathit{ID} \end{cases}$$

$$\tilde{\sigma}_5 = \tilde{\sigma}_4[\texttt{z} \mapsto \tilde{A}[\![2]\!]\tilde{\sigma}_4] \qquad \tilde{\sigma}_{10} = \tilde{\sigma}_9[\texttt{x} \mapsto \tilde{A}[\![2]\!]\tilde{\sigma}_9]$$

$$\tilde{\sigma}_6 = \begin{cases} (\tilde{\sigma}_3 \sqcup \tilde{\sigma}_5)[\texttt{flow} \mapsto \tilde{\sigma}_1(\texttt{flow})] \\ \quad \text{if } \tilde{B}[\![\texttt{x} = \texttt{1}]\!]\tilde{\sigma}_1 = \mathit{ID} \\ \tilde{\sigma}_3 \sqcup \tilde{\sigma}_5 \text{ if } \tilde{B}[\![\texttt{x} = \texttt{1}]\!]\tilde{\sigma}_1 = \mathit{NID} \end{cases} \qquad \tilde{\sigma}_{11} = \begin{cases} (\tilde{\sigma}_8 \sqcup \tilde{\sigma}_{10})[\texttt{flow} \mapsto \tilde{\sigma}_6(\texttt{flow})] \\ \quad \text{if } \tilde{B}[\![\texttt{y} = \texttt{1}]\!]\tilde{\sigma}_6 = \mathit{ID} \\ \tilde{\sigma}_8 \sqcup \tilde{\sigma}_{10} \text{ if } \tilde{B}[\![\texttt{y} = \texttt{1}]\!]\tilde{\sigma}_6 = \mathit{NID} \end{cases}$$

Figure 6. Data flow equations for the simple example program.

Solution of the Introductory Example. The Jacobi iteration of the system of flow equations in Figure 6 will reach a fixpoint after less that 10 iterations. The abstract states for two interesting program points are:

$$\tilde{\sigma}_6 = [\texttt{x} \mapsto \mathit{ID}, \texttt{y} \mapsto \mathit{NID}, \texttt{z} \mapsto \mathit{ID}, \texttt{flow} \mapsto \mathit{NID}]$$
$$\tilde{\sigma}_{11} = [\texttt{x} \mapsto \mathit{NID}, \texttt{y} \mapsto \mathit{NID}, \texttt{z} \mapsto \mathit{ID}, \texttt{flow} \mapsto \mathit{NID}]$$

The result of the analysis of the first if-statement ($\tilde{\sigma}_6$) is that z is set to *ID*. The reason for this is that the first condition `if x = 1` is input dependent (since x is). Therefore, z is regarded as input dependent (even if z is assigned constants in both edges!).

The result of the analysis of the entire example ($\tilde{\sigma}_{11}$) is that z is set to *ID* (as described above) and that x is set to *NID* due to the second condition `if y = 1` is not input dependent (since y is not). For the code generation, this means that the first if-statement must generate predicated code, while the second can generate ordinary code.

7. Find-First Example

As the second example we use the Find-First algorithm; it will calculate the index of the first occurrence of a given key in an unsorted array. We show the use of our analysis as an aid to transfer the algorithm from a traditional, average speed oriented code to a WCET-oriented coding style.

Find-First Algorithm, Traditional Solution. The Find-First algorithm coded in C as shown in Figure 7 starts the search from the first element. Upon a match between the key and an element, the algorithm terminates the search and returns the current array index. This means the algorithm has a varying execution time.

We translate this code to While code for our analysis, as shown in Figure 8. We assume that the expressions `SIZE` and `SIZE-1` represent numerals. For clarity, we allow the variables names to be longer than one character.

This program is analyzed by first setting up the flow graph and the abstract states for program points as depicted in Figure 9.

```
static int find_first_trad(int key, int a[])
{
   int i;
   int position = SIZE;
   int found = 0;

   for(i=0;
   !found && (i<=SIZE-1);
   i++) /* max. iterations: SIZE */
   {
     if (a[i] = key)
     {
       found = 1;
       position = i;
     }
   }
   return position;
}
```

Figure 7. Find-First, traditional C code.

```
position := SIZE;
found := 0;
i := 0;
while (not(found) & (i <= SIZE-1))
(
  if (a[i] = key)
  (
    found := 1;
    position := i
  )
  i = i + 1;
)
```

Figure 8. Find-First, traditional, While code.

We set the initial state to

$$\begin{aligned}\tilde{\sigma}_1 = [&\texttt{a[]} \mapsto \mathit{ID}, \texttt{position} \mapsto \mathit{NID}, \texttt{found} \mapsto \mathit{NID},\\ &\texttt{i} \mapsto \mathit{NID}, \texttt{key} \mapsto \mathit{ID}, \texttt{flow} \mapsto \mathit{NID}]\end{aligned}$$

Then we perform an iteration over these equations until the fixpoint is reached, which in this case requires about 20 iterations.

In the fixpoint, the abstract states for two important program points are:

$$\begin{aligned}\tilde{\sigma}_5 = [&\texttt{a[]} \mapsto \mathit{ID}, \texttt{position} \mapsto \mathit{ID}, \texttt{found} \mapsto \mathit{ID},\\ &\texttt{i} \mapsto \mathit{ID}, \texttt{key} \mapsto \mathit{ID}, \texttt{flow} \mapsto \mathit{NID}]\\ \tilde{\sigma}_6 = [&\texttt{a[]} \mapsto \mathit{ID}, \texttt{position} \mapsto \mathit{ID}, \texttt{found} \mapsto \mathit{ID},\\ &\texttt{i} \mapsto \mathit{ID}, \texttt{key} \mapsto \mathit{ID}, \texttt{flow} \mapsto \mathit{ID}]\end{aligned}$$

S_1 = `pos := SIZE`

S_2 = `found := 0`

S_3 = `i := 0`

S_3 = `while` (b_1) `...`

b_1 = `not(found) & (i <= SIZE-1)`

S_5 = `if` b_2 `then` S_6 `else skip;` S_7

b_2 = `a[i] = key`

S_6 = S_8; S_9

S_7 = `i:=i+1`

S_8 = `found := 1`

S_9 = `position := i`

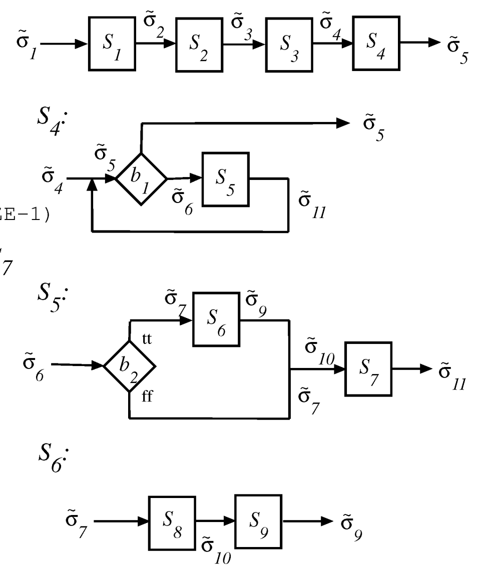


Figure 9. Flow graphs and abstract states for the Find-First example.

This result means that both conditions are input dependent, i.e.,

$$\tilde{B}[\![b_1]\!]\tilde{\sigma}_5 = \tilde{B}[\![b_2]\!]\tilde{\sigma}_6 = ID$$

Is there a way to transform the code to a less input dependent, WCET-oriented version?

Find-First Algorithm, WCET-Oriented Solution. Looking closer on the code, we conclude that the second condition

$$b_2 = \texttt{a[i] = key}$$

must always be input dependent, since it always uses the input parameters.

However, the first condition

$$b_1 = \texttt{not(found)}\ \&\ (\texttt{i} <= \texttt{SIZE - 1})$$

is input dependent only because of the test on the variable `found`. If we could remove that test, the loop could be made not input dependent.

This is exactly what has been made in the WCET-oriented version of the code, shown in Figure 10.

The WCET-oriented solution traverses the array from the last to the first element, i.e., the number of iterations is always constant. Whenever a match is found, the index is saved in the variable `position`. As the algorithm traverses the array backwards, this variable will hold the wanted index at termination.

```
position := SIZE;
i := SIZE-1;
while (i >= 0)
(
  if (a[i] = key)
  (
    position := i
  )
i := i - 1
)
```

Figure 10. Find-First, WCET-oriented While code.

Using our method to analyze this code, we find, after about 10 iterations, a fixpoint where indeed the first condition *is* input independent, i.e., *NID*.

This algorithm has three advantages over the traditional one, from a WCET point of view:

- It has a small jitter, since it always performs the same number of iterations (execution time variations may still occur due to number of matches).
- The WCET for this code is shorter than the WCET for the traditional code, since it contains fewer assignments and tests.
- Due to the reduced number of paths, it may be simpler to analyze this code for WCET as compared to traditional code.

It should be obvious how this code should be transferred to C, so we omit that code.

There is a straight-forward method to transform this WCET-oriented solution to a single-path program, as described in Puschner and Burns (2002). Only the second condition has to be translated to conditional code, as pointed out by our analysis.

8. Related Work

There is a rich literature on program analysis. We give some general references here, and one that is closely related.

Our analysis is related to classical data flow analyses used in optimizing compilers, such as reaching definitions analysis and live variable analysis. The textbook by Nielson et al. (1999) provides a thorough description of classical data flow analysis, and numerous references to the literature in the area. Also closely related is *program slicing*, which is an analysis identifying, for a given program statement, which other statements that can affect its outcome. A survey is found in Tip (1995). Most closely related, however, is *binding-time analysis*. This is an analysis used in *partial evaluation* (Sestoft and Zamulin, 1988; Jones et al., 1993). Partial evaluation is a program transformation technique, which produces simplified code for programs given that some input is known. Binding-time analysis is used to speed up this transformation: it calculates, for different program parts, if they are dependent only on known input data ("static") or possibly unknown input ("dynamic") before performing the actual code simplifications.

As a matter of fact, our analysis can be seen as a binding-time analysis albeit applied in a quite different context. Traditionally, partial evaluation and binding-time analysis have been considered mainly for declarative languages. Lately, a partial evaluator for C has been developed and was recently published (Consel et al., 2004). This partial evaluator uses an independently developed binding-time analysis that is akin to ours. However, the technical details differ, and we are not aware of any proof of its correctness.

9. Correctness Proof of the Analysis

In this section we give a proof of correctness for the program analysis. We develop a framework for reasoning about input data dependence in general, and we then prove the correctness of our analysis within this framework.

First, we must define formally what we mean by a program fragment being non-input dependent. To that end, we define a non-standard semantics for our While language that defines traces of statement executions rather than state transitions. A statement is then input data-independent whenever the subtrace of its executions is the same, regardless of the initial state of the program.

We need some notation for sequences (traces) over some alphabet. We denote the *empty trace* by Λ. $t_1 \cdot t_2$ denotes the concatenation of traces t_1 and t_2. *Projection* of trace t w.r.t. letter x, $t \upharpoonright x$, is defined by:

$$\begin{aligned} \Lambda \upharpoonright x &= \Lambda \\ (x \cdot s) \upharpoonright x &= x \cdot (s \upharpoonright x) \\ (y \cdot s) \upharpoonright x &= (s \upharpoonright x), \quad y \neq x \end{aligned}$$

Prefix ordering $\leq$ is a partial order on traces, with Λ as least element. If the set of infinite traces over the alphabet is added, then every increasing sequence has a least upper bound and the set of finite and infinite traces forms a cpo under prefix ordering. Concatenation and projection are continuous functions over this cpo, provided they are extended to infinite traces in the natural way.

In our trace semantics, the alphabet consists of the possible assignment statements. For simplicity, we assume each assignment statement only occurs once in each program. Then, each occurrence of a statement in a trace represents an execution of that very statement (and no other statement). It is easy to extend the While language with labels on assignments, and demand that statements are labeled in a way such that each statement can be uniquely identified even if occurring several times in a program.

Definition 9.1. The *trace semantics* of the While language, $\mathcal{T}[\![\ \ ]\!]$, is defined by:

$$\begin{aligned}
\mathcal{T}[\![\texttt{skip}]\!] &= \lambda\sigma.\Lambda \\
\mathcal{T}[\![var := a]\!] &= \lambda\sigma.(var := a) \\
\mathcal{T}[\![var[a_1] := a_2]\!] &= \lambda\sigma.(var[a_1] := a_2) \\
\mathcal{T}[\![S_1; S_2]\!] &= \lambda\sigma.\mathcal{T}[\![S_1]\!]\sigma \cdot \mathcal{T}[\![S_2]\!](\mathcal{S}[\![S_1]\!]\sigma) \\
\mathcal{T}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] &= \lambda\sigma.\textbf{if}\ \mathcal{B}[\![b]\!]\ \textbf{then}\ \mathcal{T}[\![S_1]\!]\sigma\ \textbf{else}\ \mathcal{T}[\![S_2]\!]\sigma \\
\mathcal{T}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] &= \mathit{fix}(\lambda g.\lambda\sigma.\textbf{if}\ \mathcal{B}[\![b]\!]\sigma\ \textbf{then}\ \mathcal{T}[\![S]\!]\sigma \cdot g(\mathcal{S}[\![S]\!]\sigma)\ \textbf{else}\ \Lambda)
\end{aligned}$$

Since the set of finite and infinite traces constitutes a cpo under prefix ordering, and since the expression in the right-hand side of the while-case is defined using continuous functions over this cpo, the least fixed point of the functional $\lambda g.(\cdots)$, mapping trace functions to trace functions, exists and is unique.

For simplicity, the trace semantics does not record the execution of conditions in `while`- and `if`-statements, but it is easily extended to do so. Also note that the trace of executing an access to an array element always is the textual access regardless of actual index value. This means that we don't distinguish between different executions of the same statement accessing different array elements. An example is executing the code

$$i := 1; \texttt{while}\ i \leq 3\ \texttt{do}\ (a[i] := i; i := i + 1)$$

which results in the subtrace $a[i] := i \cdot a[i] := i \cdot a[i] := i$ for the array assignment statement rather than $a[1] := i \cdot a[2] := i \cdot a[3] := i$. For a machine with uniform memory access times there is hardly any reason to distinguish these memory references. However, if necessary it is easy to modify the trace semantics to distinguish between array accesses going to different elements. Our correctness result holds with minor modifications also for this trace semantics.

We now define input data independence under constraints on initial states of the form "for all variables in X, the states agree on the variable values in X". First, we need some more notation. For any set of program variables X, define $\sigma|_X$ as the restriction of σ to X. (Here, we treat arrays as single program variables, so an array variable *var* might occur in these sets rather than its individual array element names $var[n]$. For such a variable *var*, $\sigma(var)$ is then a function mapping array indices to values.)

Definition 9.2. The assignment statement s is *input data independent in S relative X*, or $di(s, S, X)$, if s occurs in S and $(\mathcal{T}[\![S]\!]\sigma) \upharpoonright s = (\mathcal{T}[\![S]\!]\sigma') \upharpoonright s$ for all states σ, σ' where $\sigma|_X = \sigma'|_X$.

In particular, s can be input data independent in S relative $\emptyset$. The execution trace of s must then be the same regardless of initial state.

We now establish some local conditions for data independence, expressed as recursive conditions on substatements given constraints on input states of the form above. Later, we will make a link between these constraints and the results of the program analysis. A simple recursive strategy to decide data-independence from the result of the program analysis is easy to formulate from these results.

Definition 9.3. $X < S > Y$ iff $\forall \sigma, \sigma'. \sigma|_X = \sigma'|_X \implies (\mathcal{S}[\![S]\!]\sigma)|_Y = (\mathcal{S}[\![S]\!]\sigma')|_Y$.

Informally, $X < S > Y$ means that if we keep the values of variables in X constant before executing X, then the values of variables in Y will always be the same after executing S. The following proposition is immediate:

Proposition 1. $X \subseteq X' \wedge X < S > Y \implies X' < S > Y$. $Y' \subseteq Y \wedge X < S > Y \implies X < S > Y'$.

For any (boolean or arithmetic) expression e, $FV(e)$ denotes the set of (free) variables in e.

Theorem 9.1. *The following holds, for all s, X, and Y:*

1. $di(s, \texttt{skip}, X)$ *for any s*, $di(var := a, var := a, X)$, *and* $di(var[a_1] := a_2, var[a_1] := a_2, X)$.
2. *If* $di(s, S_1, X)$ *then* $di(s, S_1; S_2, X)$. *If* $X < S_1 > Y$ *and* $di(s, S_2, Y)$ *then* $di(s, S_1; S_2, X)$.
3. *If* $FV(b) \subseteq X$ *and* $di(s, S_1, X)$ *then* $di(s, \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, X)$ *and* $di(s, \texttt{if } b \texttt{ then } S_2 \texttt{ else } S_1, X)$.
4. *If* $FV(b) \subseteq X \cap Y$, $di(s, S, X \cap Y)$, *and* $X \cap Y < S > Y$, *then* $di(s, \texttt{while } b \texttt{ do } S, X)$.

Proof:

1. Trivial.
2. The first statement is trivial to prove, observing that $\mathcal{T}[\![S_1; S_2]\!]\sigma \restriction s = (\mathcal{T}[\![S_1]\!]\sigma \cdot \mathcal{T}[\![S_2]\!](\mathcal{S}[\![S_1]\!]\sigma)) \restriction s = \mathcal{T}[\![S_1]\!]\sigma \restriction s \cdot \mathcal{T}[\![S_2]\!](\mathcal{S}[\![S_1]\!]\sigma) \restriction s$ plus the fact that s, since it only can occur once in a program, can occur only in S_1 or S_2 but not both. The second statement in 2 also follows quite directly, from the same observation and the definition of $X < S_1 > Y$.
3. Consider the first case. Note that s must belong to S_1. When $\sigma|_X = \sigma'|_X$ and $FV(b) \subseteq X$ then $\mathcal{B}[\![b]\!]\sigma = \mathcal{B}[\![b]\!]\sigma'$. If *tt*, then $\mathcal{T}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\sigma \restriction s = \mathcal{T}[\![S_1]\!]\sigma \restriction s = \mathcal{T}[\![S_1]\!]\sigma' \restriction s = \mathcal{T}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\sigma' \restriction s$. If *ff*, then $\mathcal{T}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\sigma \restriction s = \mathcal{T}[\![S_2]\!]\sigma \restriction s = \Lambda = \mathcal{T}[\![S_2]\!]\sigma' \restriction s = \mathcal{T}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\sigma' \restriction s$. The second case is symmetric to the first.
4. Define $F(g) = \lambda\sigma.\textbf{if } \mathcal{B}[\![b]\!]\sigma \textbf{ then } \mathcal{T}[\![S]\!]\sigma \cdot g(\mathcal{S}[\![S]\!]\sigma) \textbf{ else } \Lambda$, $\theta_0 = \lambda\sigma.\Lambda$, and $\theta_i = F(\theta_{i-1})$ for $i > 0$. Then $\mathcal{T}[\![\texttt{while } b \texttt{ do } S]\!] = \bigsqcup_i \theta_i$. We first prove, by induction over i, that $di(s, \theta_i, X)$ holds for all i under the assumed conditions.[1]

- $i = 0$: $\theta_0\sigma = \Lambda$ for all σ, and the result follows immediately.

- $i > 0$: assume true for $i - 1$. When $\sigma|_{X\cap Y} = \sigma'|_{X\cap Y}$ then $\mathcal{B}[\![b]\!]\sigma = \mathcal{B}[\![b]\!]\sigma'$. If *tt*, then $\theta_i\sigma \restriction s = (\mathcal{T}[\![S]\!]\sigma \cdot \theta_{i-1}(\mathcal{S}[\![S]\!]\sigma)) \restriction s = \mathcal{T}[\![S]\!]\sigma \restriction s \cdot \theta_{i-1}(\mathcal{S}[\![S]\!]\sigma) \restriction s$. By the assumption that $di(s, S, X \cap Y)$, plus assumptions on σ, σ' we have $\mathcal{T}[\![S]\!]\sigma \restriction s = \mathcal{T}[\![S]\!]\sigma' \restriction s$, Furthermore, by Proposition 9, $X \cap Y < S > Y \implies X \cap Y < S > X \cap Y$. Thus, $\mathcal{S}[\![S]\!]\sigma|_{X\cap Y} = \mathcal{S}[\![S]\!]\sigma'|_{X\cap Y}$, and by the induction hypothesis follows that $\theta_{i-1}(\mathcal{S}[\![S]\!]\sigma) \restriction s = \theta_{i-1}(\mathcal{S}[\![S]\!]\sigma') \restriction s$. The result follows.

  When $\mathcal{B}[\![b]\!]\sigma = \mathcal{B}[\![b]\!]\sigma' = \mathit{ff}$, then $\theta_i\sigma = \theta_i\sigma' = \Lambda$ and the result follows immediately.

  Now, what about the fixed-point $\bigsqcup_i \theta_i$? For each pair of states σ, σ' such that $\sigma|_X = \sigma'|_X$ holds that the predicate $P(\theta) \equiv (\theta\sigma)\restriction s = (\theta\sigma')\restriction s$ is *inclusive* (Nielson and Nielson, 1992). Since $P(\theta_i)$ holds for each i we can, by the principle of fixed point induction (Nielson and Nielson, 1992), then conclude that $P(\bigsqcup_i \theta_i)$ holds as well. □

In the sequel we will show that the program analysis will derive sets of "data-independent" variables, with abstract value *NID* in the corresponding states, for which Theorem 9.1 holds. The remaining test for data independence is then simply to check whether, for each condition possibly affecting the statement in question, the free variables of the condition belong to this set of variables or not. If they all belong to the respective set of data-independent variables derived by the program analysis, then the statement is data-independent.

Before proceeding, we will however prove some properties of the statements of form $X < S > Y$. These properties will be useful in the proofs that follow.

Proposition 2. *The following holds:*

1. $X < \texttt{skip} > X$ *for all* X.
2. $FV(a) \subseteq X \implies X < var := a > X \cup \{var\}$.
3. $FV(a_1) \subseteq X \wedge FV(a_2) \subseteq X \implies X < var[a_1] := a_2 > X \cup \{var\}$.
4. $X < S_1 > Y \wedge Y < S_2 > Z \implies X < S_1; S_2 > Z$.
5. $FV(b) \subseteq X \wedge X < S_1 > Y \wedge X < S_2 > Y' \implies X < \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 > Y \cap Y'$.
6. $FV(b) \subseteq X \cap Y \wedge X \cap Y < S > Y \implies X < \texttt{while}\ b\ \texttt{do}\ S > X \cap Y$.

Proof: The results are proven one by one using the standard denotational semantics for the While language. The proofs for Case 1 is immediate. For Case 2 it follows since then $\mathcal{A}[\![a]\!]\sigma = \mathcal{A}[\![a]\!]\sigma'$ for all σ, σ' such that $\sigma|_X = \sigma'|_X$. Case 3 is similar, only that it has to be ensured that both $\mathcal{A}[\![a_1]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma'$ and $\mathcal{A}[\![a_2]\!]\sigma = \mathcal{A}[\![a_2]\!]\sigma'$ so that the array value of *var* is the same after the assignment.

For Case 4, assume that $\sigma|_X = \sigma'|_X$. Then $\mathcal{S}[\![S_1]\!]\sigma|_Y = \mathcal{S}[\![S_1]\!]\sigma'|_Y$. By $Y < S_2 > Z$ then follows that $\mathcal{S}[\![S_2]\!](\mathcal{S}[\![S_1]\!]\sigma)|_Z = \mathcal{S}[\![S_2]\!](\mathcal{S}[\![S_1]\!]\sigma')|_Z$, that is: $\mathcal{S}[\![S_1; S_2]\!]\sigma|_Z = \mathcal{S}[\![S_1; S_2]\!]\ \sigma'|_Z$, which proves the result.

In Case 5, we have the cases $\mathcal{B}[\![b]\!]\sigma = \mathcal{B}[\![b]\!]\sigma' = tt$ and ff, respectively. In the first case, $\mathcal{S}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\sigma = \mathcal{S}[\![S_1]\!]\sigma$ and $\mathcal{S}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\sigma' = \mathcal{S}[\![S_1]\!]\sigma'$, which from $X < S_1 > Y$ yields $(\mathcal{S}[\![\texttt{if } b \texttt{ then } S_b \texttt{ else } S_2]\!]\sigma)|_Y = (\mathcal{S}[\![\texttt{if } b \texttt{ then } S_b \texttt{ else } S_2]\!]\sigma')|_Y$. Similarly, if $\mathcal{B}[\![b]\!]\sigma = \mathcal{B}[\![b]\!]\sigma' = ff$, we obtain $(\mathcal{S}[\![\texttt{if } b \texttt{ then } S_b \texttt{ else } S_2]\!]\sigma)|_{Y'} = (\mathcal{S}[\![\texttt{if } b \texttt{ then } S_b \texttt{ else } S_2]\!]\sigma')|_{Y'}$. In any case holds that $(\mathcal{S}[\![\texttt{if } b \texttt{ then } S_b \texttt{ else } S_2]\!]\sigma)|_{Y\cap Y'} = (\mathcal{S}[\![\texttt{if } b \texttt{ then } S_b \texttt{ else } S_2]\!]\sigma')|_{Y\cap Y'}$.

Finally, in Case 6, we use fixed point induction in the same manner as in the proof of Theorem 9.1 (except that this proof concerns the standard semantics, not the trace semantics). Denote the partial state transition functions successively approximating $\mathcal{S}[\![\texttt{while } b \texttt{ do } S]\!]$ by γ_i, $i \geq 0$. We have $\gamma_0 = \lambda\sigma.\bot$ (undefined for any state), and $\gamma_i = \lambda\sigma.$ **if** $\mathcal{B}[\![b]\!]\sigma$ **then** $\gamma_{i-1}(\mathcal{S}[\![S]\!]\sigma)$ **else** σ. Assume that $FV(b) \subseteq X \cap Y$ and $X \cap Y < S > Y$. First we show, by induction over i, that for all i holds that $X \cap Y < \theta_i > X \cap Y$ (which implies $X < \theta_i > X \cap Y$ for all i):

- $i = 0$: trivially true (since θ_0 maps no state to any state).

- $i > 0$: assume true for $i - 1$. Consider σ, σ' such that $\sigma|_{X\cap Y} = \sigma'|_{X\cap Y}$. Then $\mathcal{B}[\![b]\!]\sigma = \mathcal{B}[\![b]\!]\sigma'$. If tt, then $\gamma_i\sigma = \gamma_{i-1}(\mathcal{S}[\![S]\!]\sigma)$ and $\gamma_i\sigma' = \gamma_{i-1}(\mathcal{S}[\![S]\!]\sigma')$. From $X \cap Y < S > Y$ we have $X \cap Y < S > X \cap Y$ (from Proposition 1): thus $(\mathcal{S}[\![S]\!]\sigma)|_{X\cap Y} = (\mathcal{S}[\![S]\!]\sigma')|_{X\cap Y}$. By the induction hypothesis, we then obtain $\gamma_{i-1}(\mathcal{S}[\![S]\!]\sigma)|_{X\cap Y} = \gamma_{i-1}(\mathcal{S}[\![S]\!]\sigma')|_{X\cap Y}$, which implies the result.

  If $\mathcal{B}[\![b]\!]\sigma = \mathcal{B}[\![b]\!]\sigma' = ff$, then $\gamma_i\sigma = \sigma$ and $\gamma_i\sigma' = \sigma'$, which immediately yields the result also in this case.

  Finally, we observe that for any σ, σ' such that $\sigma|_{X\cap Y} = \sigma'|_{X\cap Y}$, the predicate is inclusive. This means that it holds also in the limit. We obtain $X \cap Y <$ $\texttt{while } b \texttt{ do } S > X \cap Y$, which implies $X < \texttt{while } b \texttt{ do } S > X \cap Y$. □

Some interesting properties are easiest expressed using *kill sets* of program variables possibly assigned in a program segment. (This also shows the relation between our analysis and classical data flow analyses (Nielson et al., 1999), which sometimes use similar kill sets.)

Definition 9.4. For any program S, its *kill set* $kill(S)$ is defined by:

$$\begin{aligned} kill(\texttt{skip}) &= \emptyset \\ kill(var := a) &= \{var\} \\ kill(var[a_1] := a_2) &= \{var\} \end{aligned}$$

$$\begin{aligned} \mathit{kill}(S_1; S_2) &= \mathit{kill}(S_1) \cup \mathit{kill}(S_2) \\ \mathit{kill}(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2) &= \mathit{kill}(S_1) \cup \mathit{kill}(S_2) \\ \mathit{kill}(\texttt{while } b \texttt{ do } S) &= \mathit{kill}(S) \end{aligned}$$

Denote the complement of the set X by $C(X)$.

Lemma 9.1. *For all programs S, and states σ such that $\mathcal{S}[\![S]\!]\sigma$ is defined, holds that $\sigma|_{C(\mathit{kill}(S))} = (\mathcal{S}[\![S]\!]\sigma)|_{C(\mathit{kill}(S))}$.*

Proof: By induction over program structure.

- $S = \texttt{skip}$, $S = \mathit{var} := a$, $S = \mathit{var}[a_1] := a_2$: immediate. (In the two latter cases, since no aliasing of variables can occur, no other variable than *var* can be touched by the respective assignment.)

- $S = S_1; S_2$: assume true for S_1 and S_2. We then have $\sigma|_{C(\mathit{kill}(S_1))} = (\mathcal{S}[\![S_1]\!]\sigma)|_{C(\mathit{kill}(S_1))}$, which implies $\sigma|_{C(\mathit{kill}(S_1))\cap C(\mathit{kill}(S_2))}(\mathcal{S}[\![S_1]\!]\sigma)|_{C(\mathit{kill}(S_1))\cap C(\mathit{kill}(S_2))}$. We also have $(\mathcal{S}[\![S_1]\!]\sigma)|_{C(\mathit{kill}(S_2))} = (\mathcal{S}[\![S_2]\!](\mathcal{S}[\![S_1]\!]\sigma))|_{C(\mathit{kill}(S_2))}$, which implies $(\mathcal{S}[\![S_1]\!]\sigma)|_{C(\mathit{kill}(S_2))\cap C(\mathit{kill}(S_1))} = (\mathcal{S}[\![S_2]\!](\mathcal{S}[\![S_1]\!]\sigma))|_{C(\mathit{kill}(S_2))\cap C(\mathit{kill}(S_1))}$. Since $C(X) \cap C(Y) = C(X \cup Y)$ it follows that $\sigma|_{C(\mathit{kill}(S_1)\cup \mathit{kill}(S_2))} = (\mathcal{S}[\![S_2]\!](\mathcal{S}[\![S_1]\!]\sigma))|_{C(\mathit{kill}(S_1)\cup \mathit{kill}(S_2))}$.

- $S = \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2$: If $\mathcal{B}[\![b]\!]\sigma = \mathit{tt}$, then $\sigma|_{C(\mathit{kill}(S_1))} = (\mathcal{S}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\sigma)|_{C(\mathit{kill}(S_1))}$. If $\mathcal{B}[\![b]\!]\sigma = \mathit{ff}$, then $\sigma|_{C(\mathit{kill}(S_1))} = (\mathcal{S}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\sigma)|_{C(\mathit{kill}(S_2))}$. In any case, it holds that $\sigma|_{C(\mathit{kill}(S_1))\cap C(\mathit{kill}(S_2))} = (\mathcal{S}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\sigma)|_{C(\mathit{kill}(S_1))\cap C(\mathit{kill}(S_2))}$, i.e., $\sigma|_{C(\mathit{kill}(S_1)\cup \mathit{kill}(S_2))} = (\mathcal{S}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\sigma)|_{C(\mathit{kill}(S_1)\cup \mathit{kill}(S_2))}$.

- $S = \texttt{while } b \texttt{ do } S$: proof by fixed-point induction, assuming true for S.

— $i = 0$: immediately true for γ_0, since $\gamma_0\sigma$ is undefined for all σ.

— $i > 0$: assume true for $i - 1$. If $\mathcal{B}[\![b]\!]\sigma = \mathit{tt}$, then $\gamma_i\sigma = \gamma_{i-1}(\mathcal{S}[\![S]\!]\sigma)$. By assumption, $\sigma|_{C(\mathit{kill}(S))} = (\mathcal{S}[\![S]\!]\sigma)|_{C(\mathit{kill}(S))}$. By the induction hypothesis, we have $((\mathcal{S}[\![S]\!]\sigma)|_{C(\mathit{kill}(S))})|_{C(\mathit{kill}(S))} = (\gamma_{i-1}(\mathcal{S}[\![S]\!]\sigma)|_{C(\mathit{kill}(S))})|_{C(\mathit{kill}(S))}$. This simplifies to $(\mathcal{S}[\![S]\!]\sigma)|_{C(\mathit{kill}(S))} = \gamma_{i-1}(\mathcal{S}[\![S]\!]\sigma)|_{C(\mathit{kill}(S))}$. We obtain the result for all states σ such that $\mathcal{B}[\![b]\!]\sigma = \mathit{tt}$. If $\mathcal{B}[\![b]\!]\sigma = \mathit{ff}$, then $\gamma_i\sigma = \sigma$ and also in this case the result follows. Again, we can note that the predicate is inclusive, which by the above implies that it holds also for the limit. □

Proposition 3. *For all programs S which terminate for all input states, and all sets of variables X, holds that $X < S > X \setminus \mathit{kill}(S)$.*

Proof: By Lemma 9.2, we then have $\sigma|_{C(\mathit{kill}(S))} = (\mathcal{S}[\![S]\!]\sigma)|_{C(\mathit{kill}(S))}$ for all σ. This implies $(\sigma|_X)|_{C(\mathit{kill}(S))} = ((\mathcal{S}[\![S]\!]\sigma)|_X)|_{C(\mathit{kill}(S))}$. Similarly, $(\sigma'|_X)|_{C(\mathit{kill}(S))} =$

$((\mathcal{S}[\![S]\!]\sigma')|_X)|_{C(kill(S))}$. When $\sigma|_X = \sigma'|_X$ we obtain $((\mathcal{S}[\![S]\!]\sigma)|_X)|_{C(kill(S))} = ((\mathcal{S}[\![S]\!]\sigma')|_X)|_{C(kill(S))}$, that is: $(\mathcal{S}[\![S]\!]\sigma)|_{X\setminus kill(S)} = (\mathcal{S}[\![S]\!]\sigma')|_{X\setminus kill(S)}$. □

Let us now turn to the abstract semantics for the While language. The abstract states map program variables to $\{ID, NID\}$. For any abstract state $\tilde{\sigma}$ we define the set of data-independent variables as $nvar(\tilde{\sigma}) = \{x \mid \tilde{\sigma}(x) = NID\}$. Abstract states can be represented by such sets of variables, and all operations on abstract states carry over to set operations. For instance, $nvar(\tilde{\sigma}_1 \sqcup \tilde{\sigma}_2) = nvar(\tilde{\sigma}_1) \cap nvar(\tilde{\sigma}_2)$. Also, rather than computing a least fixed point for abstract states, one can compute a greatest fixed point (w.r.t. set inclusion) for the variable sets.

Lemma 9.2. *For all programs S that terminate for all inputs, and abstract states $\tilde{\sigma}$ such that $\tilde{\sigma}(flow) = ID$, holds that $(\tilde{\mathcal{S}}[\![S]\!]\tilde{\sigma})(flow) = ID$ and $nvar(\tilde{\mathcal{S}}[\![S]\!]\tilde{\sigma}) = nvar(\tilde{\sigma}) \setminus kill(S)$.*

Proof: Note that $\tilde{\mathcal{A}}[\![a]\!]\tilde{\sigma} = \tilde{\mathcal{B}}[\![b]\!]\tilde{\sigma} = ID$ as soon as $\tilde{\sigma}(flow) = ID$. It is then easy to see that the first statement holds, by a simple inspection of the different cases. (Sequencing requires a trivial induction step. For while, the case $\tilde{\mathcal{B}}[\![b]\!](\tilde{\mathcal{S}}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]\tilde{\sigma}) = NID$ yields a contradiction.)

The second statement holds trivially for `skip`, as well as both types of assignments. For sequencing it follows by a simple induction on S_1 and S_2, noticing that by induction $\tilde{\sigma}(flow) = NID \Longrightarrow (\tilde{\mathcal{S}}[\![S_1]\!]\tilde{\sigma})(flow) = NID$. For if-then-else, the result again follows by a simple induction on S_1 and S_2. For while, finally, we must have the case $\tilde{\mathcal{B}}[\![b]\!](\tilde{\mathcal{S}}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]\tilde{\sigma}) = ID$: then $\tilde{\mathcal{S}}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]\tilde{\sigma}$ is the least solution to the equation $w = (\tilde{\sigma} \sqcup \tilde{\mathcal{S}}[\![S]\!]w[flow \mapsto ID])[flow \mapsto w(flow)]$. This equation can be simplified to $w = \tilde{\sigma} \sqcup \tilde{\mathcal{S}}[\![S]\!]w$. The corresponding equation for variable sets is $nvar(w) = nvar(\tilde{\sigma}) \cap nvar(\tilde{\mathcal{S}}[\![S]\!]w)$: the greatest fixed point for this equation corresponds to the least fixed point of the original equation. By induction on S, we have $nvar(\tilde{\mathcal{S}}[\![S]\!]w) = nvar(w) \setminus kill(S)$. We obtain the equation $nvar(w) = nvar(\tilde{\sigma}) \cap (nvar(w) \setminus kill(S))$. It is easy to see that the greatest fixed point to this equation indeed is $nvar(w) = nvar(\tilde{\sigma}) \setminus kill(S)$. □

We are finally in a position to state our main result, which links the abstract values calculated by our analysis to statements that can be used, through Theorem 9.1, to prove input data independence of statements.

Theorem 9.2. *For all abstract states $\tilde{\sigma}$ and terminating programs S, it holds that $nvar(\tilde{\sigma}) < S > nvar(\tilde{\mathcal{S}}[\![S]\!]\tilde{\sigma})$.*

Proof: Note that by Proposition 3 and Lemma 9.2, we have already proved the case when $\tilde{\sigma}(flow) = ID$. Thus, assume $\tilde{\sigma}(flow) = NID$ in the sequel.

The proof is by structural induction over programs. `skip` is trivial. For assignments $var{:=}a$, if $\tilde{\mathcal{A}}[\![a]\!]\tilde{\sigma} = NID$ then $FV(a) \subseteq nvar(\tilde{\sigma})$ and $nvar(\tilde{\mathcal{S}}[\![var := a]\!]\tilde{\sigma}) = nvar(\tilde{\sigma}) \cup \{var\}$. The result then follows from Proposition 2. Otherwise, $nvar(\tilde{\mathcal{S}}[\![var := a]\!]\tilde{\sigma}) = nvar(\tilde{\sigma}) \setminus \{var\}$, and the result follows from Proposition 3. The case for array assignment is similar to the case for ordinary assignments.

For $S_1 ; S_2$, assume $nvar(\tilde{\sigma}) < S_1 > nvar(\tilde{\mathcal{S}}[\![S_1]\!]\tilde{\sigma})$ and $nvar(\tilde{\sigma}) < S_2 > nvar(\tilde{\mathcal{S}}[\![S_2]\!]\tilde{\sigma})$ for all $\tilde{\sigma}$. With $\tilde{\sigma}$ replaced by $\tilde{\mathcal{S}}[\![S_1]\!]\tilde{\sigma}$ in the second statement, we obtain $nvar(\tilde{\mathcal{S}}[\![S_1]\!]\tilde{\sigma}) < S_2 > nvar(\tilde{\mathcal{S}}[\![S_2]\!](\tilde{\mathcal{S}}[\![S_1]\!]\tilde{\sigma}))$. Proposition 2 then yields the result.

Similarly, for `if` b `then` S_1 `else` S_2, assume the same for S_1 and S_2. When $FV(b) \subseteq nvar(\tilde{\sigma})$ we have $\tilde{\mathcal{B}}[\![b]\!]\tilde{\sigma} = NID$ and $\tilde{\mathcal{S}}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\tilde{\sigma} = \tilde{\mathcal{S}}[\![S_1]\!]\tilde{\sigma} \sqcup \tilde{\mathcal{S}}[\![S_2]\!]\tilde{\sigma}$, that is: $nvar(\tilde{\mathcal{S}}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\tilde{\sigma}) = nvar(\tilde{\mathcal{S}}[\![S_1]\!]\tilde{\sigma}) \cap nvar(\tilde{\mathcal{S}}[\![S_2]\!]\tilde{\sigma})$. By induction, $nvar(\tilde{\sigma}) < S_1 > nvar(\tilde{\mathcal{S}}[\![S_1]\!]\tilde{\sigma})$ and $nvar(\tilde{\sigma}) < S_2 > nvar(\tilde{\mathcal{S}}[\![S_2]\!]\tilde{\sigma})$. The result then follows by Proposition 2.

If $FV(b) \not\subseteq nvar(\tilde{\sigma})$ then $\tilde{\mathcal{B}}[\![b]\!]\tilde{\sigma} = ID$, and $\tilde{\mathcal{S}}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\tilde{\sigma} = (\tilde{\mathcal{S}}[\![S_1]\!]\tilde{\sigma}[flow \mapsto ID] \sqcup \tilde{\mathcal{S}}[\![S_2]\!]\tilde{\sigma}[flow \mapsto ID])[flow \mapsto \tilde{\sigma}(flow)]$. By Lemma 9.2, $nvar(\tilde{\mathcal{S}}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\tilde{\sigma})$ equals $(nvar(\tilde{\sigma}[flow \mapsto ID]) \setminus kill(S_1) \cap nvar(\tilde{\sigma}[flow \mapsto ID]) \setminus kill(S_2))[flow \mapsto \tilde{\sigma}(flow)]$. Since $flow \notin kill(S)$ for any S, this equals $nvar(\tilde{\sigma}) \setminus kill(S_1) \cap nvar(\tilde{\sigma}) \setminus kill(S_2) = nvar(\tilde{\sigma}) \setminus (kill(S_1) \cup kill(S_2))$. The result then follows from Proposition 3.

Finally consider `while` b `do` S. If $\tilde{\mathcal{B}}[\![b]\!]\tilde{\sigma} = ID$, then $nvar(\tilde{\mathcal{S}}[\![\texttt{while } b \texttt{ do } S]\!]) = nvar(\tilde{\sigma}) \setminus kill(S)$ can be proved in the same way as in the proof of Lemma 9.2. The result then follows by Proposition 3. If $\tilde{\mathcal{B}}[\![b]\!]\tilde{\sigma} = NID$, then $\tilde{\mathcal{S}}[\![\texttt{while } b \texttt{ do } S]\!]\tilde{\sigma}$ is given by the least fixed point to the equation $w = \tilde{\sigma} \sqcup \tilde{\mathcal{S}}[\![S]\!]w$, or for sets of variables, the greatest fixed point to $nvar(w) = nvar(\tilde{\sigma}) \cap nvar(\tilde{\mathcal{S}}[\![S]\!]w)$. By induction on S, $nvar(w) < S > nvar(\tilde{\mathcal{S}}[\![S]\!]w)$. Replacing $nvar(w)$, we obtain $nvar(\tilde{\sigma}) \cap nvar(\tilde{\mathcal{S}}[\![S]\!]w) < S > nvar(\tilde{\mathcal{S}}[\![S]\!]w)$. Using Proposition 9 twice, on each side, we obtain $nvar(\tilde{\sigma}) < S > nvar(\tilde{\sigma}) \cap nvar(\tilde{\mathcal{S}}[\![S]\!]w)$, that is: $nvar(\tilde{\sigma}) < S > nvar(w)$, which proves the result. □

10. Conclusions and Future Work

In this paper we described an analysis technique that identifies input-data dependent control conditions in loops or branching statements of real-time code. This analysis is needed for two purposes.

First, tools based on the analysis support the programmer in writing code that avoids input-data dependent control flow as far as possible.

Second, the single-path conversion technique relies on this analysis to remove input-data dependent control dependencies from the code. Both, avoiding and eliminating input-data dependent control flow are important to produce good real-time code, i.e., code with a small execution-time jitter for which safe and tight upper bounds on the execution-time can be computed.

The analysis of input dependencies is based on abstract interpretation. The **While** language was used to provide a formal description of the abstract-interpretation framework and demonstrate how it finds input dependencies. A number of examples were used to illustrate the approach.

As a next step we plan to extend the simple semantics used in this paper to a real imperative programming language like C.

Further, we will develop a tool that implements the analysis and analyzes real code. The tool will be realized as a part of a compiler.

Note

1. Somewhat improperly, we use a semantic function from states to traces rather than a While program in the statement to be proved, but the meaning should be clear.

References

Allen, J., Kennedy, K., Porterfield, C., and Warren, J. 1983. Conversion of Control Dependence to Data Dependence. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189.

Cousot, P., and Cousot, R. 1977. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pp. 238–252.

Consel, C., Lawall, J. L., and Le Meur, A.-F. 2004. A tour of Tempo: A program specializer for the C language. *Science of Computer Programming*, 52(1–3):341–370.

Fauster, J., Kirner, R., and Puschner, P. 2003. Intelligent Editor for Writing WCET-Oriented Programs. In *Proc. 3rd International Conference on Embedded Software (EMSOFT'03)*, pp. 190–205.

Gustafsson, J. 2000. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation.* PhD thesis, Department of Computer Systems, Information Technology, Uppsala University.

Jones, N. D., Gomard, C. K., and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation.* Hertfordshire, UK: Prentice Hall.

Mahlke, S., Hank, R., McCormick, J., August, D., and Hwu, W. 1995. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In *Proc. 22nd International Symposium on Computer Architecture*, pp. 138–150.

Nielson, H. R. and Nielson, F. 1992. *Semantics with Applications*. John Wiley & Sons.

Nielson, F., Nielson, H. R., and Hankin, C. 1999. *Principles of Program Analysis*, Springer. ISBN: 3-540-65410-0.

Puschner, P. and Burns, A. 2002. Writing temporally predictable code. In *Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pp. 85–91.

Puschner, P. 2002. Transforming execution-time boundable code into temporally predictable code. In B. Kleinjohann, K.H. (Kane) Kim, L. Kleinjohann, and A. Rettberg, editors, *Design and Analysis of Distributed Embedded Systems*, Kluwer Academic Publishers, pp. 163–172.

Puschner, P. 2003. Algorithms for dependable hard real-time systems. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pp. 26–31.

Sestoft, P. and Zamulin, A. V. 1988. Annotated bibliography on partial evaluation and mixed computation. *New Generation Computing*, pp. 309–354.

Tip, F. 1995. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189.

Raimund Kirner is an assistant professor in computer science in the Real-Time Systems Group of the Vienna University of Technology. He received a Master's degree in computer science and a doctoral degree in technical sciences both from the Vienna University of Technology in Austria in the years 2000 and 2003, respectively. His research interests include worst-case execution time analysis, compiler support for worst-case execution time analysis, and the verification of real-time systems.

Peter Puschner is a professor in computer science at Vienna University of Technology. His main research focus is on worst-case execution time (WCET) analysis for real-time programs. Puschner has been working on WCET analysis for more than ten years and has strongly influenced the state of the art in this field. He has published numerous papers on WCET analysis and software/hardware architectures supporting temporal predictability. He was a guest editor for the special issue on WCET analysis of the Kluwer International Journal on Real-Time Systems and chaired the program committee of the IEEE International Symposium on Object-oriented Real-time distributed Computing in 2003 and the Euromicro Real-Time Systems Conference in 2004. In 2000/2001 Peter Puschner spent one year as a Marie-Curie research fellow at the University of York, England.