Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 203 (2009) 71–85
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

WCET Analysis of Data Dependent, Component Oriented, Embedded Software Systems

Peter Szulman[1]

Software Engineering Group
FZI Research Center for Information Technologies
Karlsruhe, Germany

Abstract

Component oriented software construction approaches have gained enormous attention in recent years. A developer has to consider functional and non-functional requirements of a customer or a market segment. For real-time systems non-functional properties - such as reliability, memory consumption and performance - are at least as important as functional requirements. In our work, we focus on the prediction of worst case execution time. In this paper we deal with the problem of analyzing data flow - control flow dependencies between components with the goal of getting tight and safe WCET predictions. We use abstract interpretation for this purpose. In this paper we show how this approach can be applied to an example taken from the domain of programmable logic controllers.

Keywords: worst case execution time, abstract interpretation, components with data dependency, programmable logic controller

1 Introduction

Component oriented software construction approaches have gained enormous attention in recent years. Building large software from pieces of pre-built, plug-in compatible software components usually lead to shorter development cycles. Several industry standards were established during the previous years. One can think of EJB, CORBA, .NET or Web services infrastructures just to mention a few. However component orientation is not only an approach owned by the domain of enterprise systems, embedded systems also profit a lot from component orientation. In our work we focus on a special domain, specifically Programmable Logic Controllers. A PLC is a small computer used for automation of industrial processes such as control of factory assembly lines. PLCs are designed to work in environments with

[1] Email: szulman@fzi.de

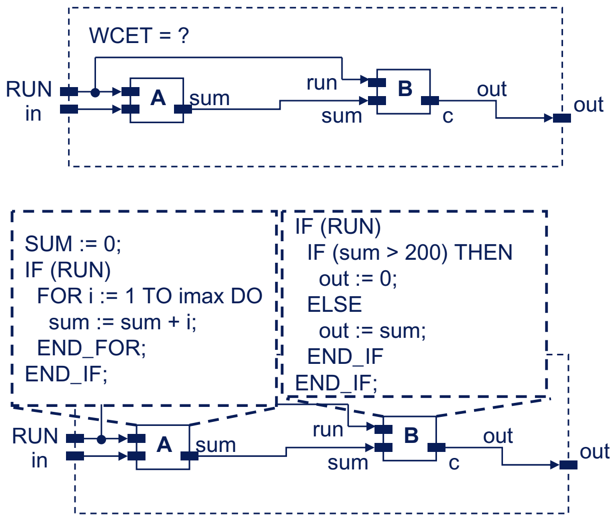
doi:10.1016/j.entcs.2009.03.027

extended temperature ranges and are resistant to electrical noise and vibration. A PLC is an example of a real-time system because it has to respond to an input condition within a strictly bounded time. Today's modern PLCs can be programmed with several programming languages as described in the standard IEC61131-3 [21]. Programmers usually make high reuse of pre-fabricated libraries containing several components.

Software development is usually triggered by the requirements of a user or by the needs of a market segment. These are mostly functional requirements. Non-functional requirements like performance, memory consumption or reliability play only a secondary role in general. However if we think of embedded systems, non-functional properties of the system are at least as important as functional ones. Instead of relying on a general purpose prediction model with the goal of verification of arbitrary non-functional properties of arbitrary component systems we focus (1) on a single non-functional property: worst case execution time (WCET) (2) and a single domain: on PLCs mentioned above (3) and develop a prediction model working as fast and delivering WCET-estimates as tight as possible.

Verification of non-functional properties of the system is a necessary, but usually hard task. The main difficulty originates from the fact that WCET is influenced by several factors: (a) The software runs on hardware. Today's PLCs differ in their CPU-architectures implementing different caching and speculative execution strategies. These kinds of "low level" features of the hardware influence the execution time enormously. (b) Usually the system has interactions with its environment: with a user or with the process itself which the PLC controls. In the PLC domain the user interactions are usually limited to starting and stopping of the control system or signaling some alarm situations. (c) Besides the previous influence factors, the structure of the software itself affects its own execution time. There are several difficulties that we have to overcome, such as, (c.1) complex control flow inside components; (c.2) data flow-control flow dependencies between components; (c.3) components with state-control flow dependencies; (c.4) or even heterogenous systems containing a mixture of white-box components with known internal structure and black-box components with hidden inside. In this paper, we address only problems in connection with complex component control flow and data flow-control flow dependencies between components (c.1 and c.2 respectively). In Figure 1, a simplified example is shown. It depicts a system having two components A and B. The (worst case) execution time of the system and respectively of component A depends on the input values *run* and *imax*. Furthermore, the run-time behavior of B is influenced by its data context, in this case by the outputs of A. In this example the internal control flow of the components is quite easy to analyze. However, there are cases with more complex control flow, for example, different types of cycles or mutually exclusive paths - just to mention a few. See Figure 2 for some simplified examples.

Worst case execution time prediction of software systems is not a brand new topic: there are several techniques already. WCET prediction is usually divided into two parts: *program path analysis* [1,11,8,10,27], which determines the sequence



```
SUM := 0;
IF (RUN)
  FOR i := 1 TO imax DO
    sum := sum + i;
  END_FOR;
END_IF;
```

```
IF (RUN)
  IF (sum > 200) THEN
    out := 0;
  ELSE
    out := sum;
  END_IF
END_IF;
```

Fig. 1. Components with data-control flow dependency

```
FOR j := 1 TO ((N+3)^2 + N*15)/2 DO
   FOR j := i TO 2 * i DO
     ..
   END_FOR;
END_FOR
```

```
IF (k = TRUE) N := 100
ELSE N:= 140;
..
FOR i := 1 TO N DO
  ..
END_FOR
```

```
IF (a == TRUE) THEN
   b := FALSE;
   ..
END_IF
..
IF (b = TRUE)
THEN
   ..
END_IF;
```

Fig. 2. Examples of complex control flow

of instructions to be executed in the worst case scenario, and *low level modeling* [6], which deals with the underlying hardware systems and computes the WCET of a known sequence of instructions. Both approaches are important in determining tight WCETs. However, these approaches do not consider component orientation and related problems like those described above. A second family of prediction approaches focus on component orientation and verification of some non-functional properties [15,16,26,36,31,32,20]. As argued above, the WCET of a system highly depends on its environment and its internal structure. In our ongoing research, we try to profit from both research communities and develop a prediction model yielding *tight* WCET estimates considering as far as possible factors influencing the WCET on today's PLCs.

This paper is organized as follows: In Section 2 we give an overview of our approach. In the main part of this paper (Section 3), we focus on one step of our approach in more detail: we describe how data-control flow dependent white bow components can be treated. Finally, we present related work and our future steps (Section 3).

2 Overview of WCET Analysis

The major goal of the analysis is to compute *safe* (no overestimation) and *tight* bounds for the worst case execution time of the system. The starting point of the analysis is the PLC program itself. Today's PLC programs are written according to a specification described in IEC61131 [17]. The current (second) edition of the standard defines two graphical and two textual PLC programming languages. (1) A *Function Block Diagram* (FBD) is a graphical language, very similar to a graphical view of a component system. An FBD describes a function block (*component*) between input and output variables (*ports*). Function blocks differ in their complexity: they range from elementary function blocks like MIN, MAX, SIN, COS to function blocks like "XYFactoryControl" responsible for the control of a whole factory. It may be *composed* from a set of finer grained function blocks. Input and output variables of different function blocks are connected by connection lines. A function block is either *black-box* (hidden inside) or *white-box*, if its refinement is available. (2) *Structured Text* (ST) is a high level, text based, block structured language (e.g iteration loops, conditional execution e.g.), similar to PASCAL. (3) The ladder logic is a method of drawing electrical logic schematics. It was originally invented to describe logic made from relays. A program in ladder logic, also called a *Ladder Diagram*, is similar to a schematic for a set for relay circuits. It is a graphical, rule-based rather, than a procedural programming language. (4) The *Instruction List* (IL) is a text based low level language and resembles assembler. (5) *A Sequential Function Chart* (SFC) is a graphical programming language. It can be used to organize programs for sequential and parallel control processing.

A PLC program can be implemented by mixing the 5 languages desribed above. For instance, an FBD describes a high level component overview of the system, where components are implemented either as ST or refined by further FBD-s. In practise languages are often paired (FBD & ST or FBD & LD). We assume in our work, that a PLC program is implemented as a mixture of ST and FBD, other languages (IL, SFC, LD) are not considered yet. Figure 1 depicts a simple PLC program written in FBD and ST.

As also shown in Figure 3, our approach consists of several steps :

(i) First we analyze the system's component structure (FBD) together with the source code (written in ST) of white box components, if available. ST source is analyzed with well known compiler techniques (syntactical and semantical analysis respectively) with the goal of producing a control flow graph for each component. Figure 4 depicts two control flow graphs extracted from the components of our example scenario 1. (The meaning of the lines having the form $x_i := x_i + 1$ is described later on.) The component CFG-s can be composed, since according to the specification we know in which order components are executed. This step yields a control flow graph for the whole system.

(ii) Given a CFG we use the *Implicite Path Enumeration Technique* IPET[24] to find the WCET of the system. With this method, we model the control flow by an integer linear program, so that the solution to the objective function is the

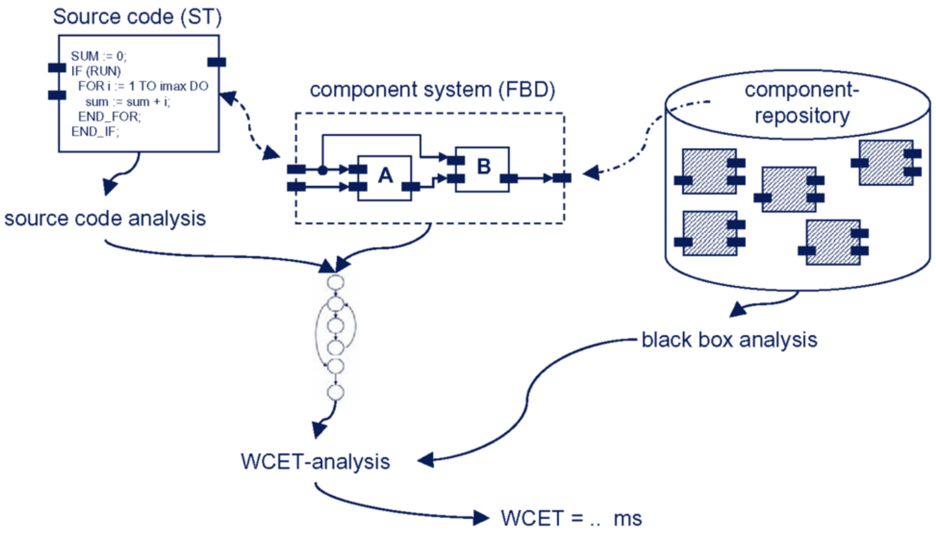


Fig. 3. Overview of our approach

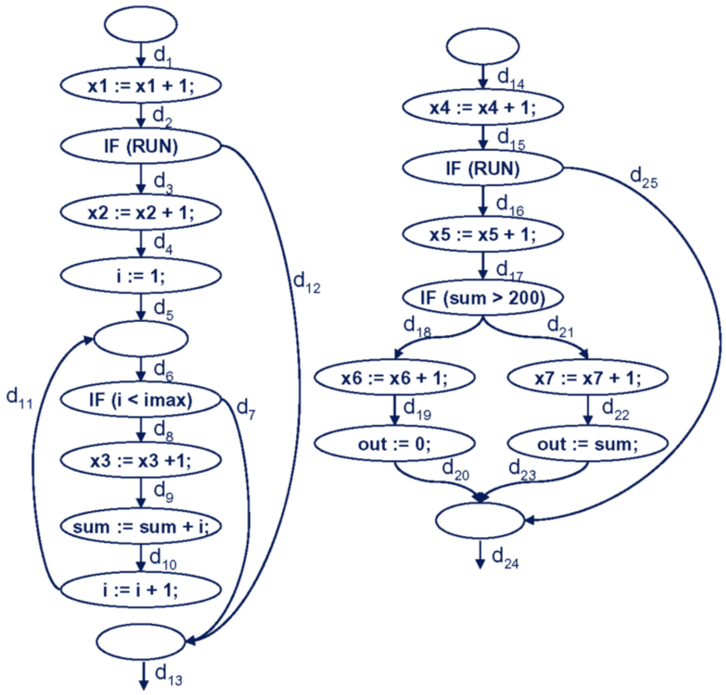


Fig. 4. Instrumented control flow graphs of the components

predicted WCET of the system. With IPET it's not necessary to enumerate all possible paths. In our current approach we assume, that each node N_i in the CFG takes a constant time c_i to execute. Let x_i be the number of times node N_i is executed on a particular path. Then equation 1 computes us the execution time of this path. The WCET of the system is given by the maximum value of this expression. Clearly, without any additional constraints on the x_i-s the maximum value of this sum would be ∞.

$$\sum_{i}^{N} x_i * c_i \tag{1}$$

There are two kinds of constraints. *Structural constraints* can be derived from the structure of the CFG directly. d_i denotes the execution count of an edge i. (We label the edges by d_i-s in Figure 4). Equation 2 says that whenever component A is called, control flows as follows: edge d_1; node x1:=x1 + 1; edge d_2.

$$x_1 = d_1 = d_2 \tag{2}$$

The following equation states that the header of the conditional *IF (RUN)* is executed as many times as the following *THEN* and *ELSE* branches combined.

$$d_2 = d_3 + d_{12} \tag{3}$$

The second family of constraints are called *functionality constraints* describing additional information such as loop execution bounds or mutually exclusive paths. We provide the functional constraints during the step *iii*. The structural constraints together with the functionality constraints and the sum to maximize (equation 1) form an *integer linear programming problem* (ILP) which can be solved by an ILP-solver.

(iii) To get a tight and safe WCET estimation, we investigate the control flow further. This step is described in the following section in more detail.

(iv) A PLC system is not composed based only on white box components. Sooner or latter, a developer will use some components provided in libraries. Some of them may come from 3'rd party libraries. Of course to be able to make tight and safe WCET estimation in a heterogenous system consisting of a mixture of white box and black box components, we assume that the 3'rd party provider also supplies context dependent execution time properties with his black-box components delivered. However, this paper does not focus on describing how this information may look like and how it may be reused.

3 Analysis of Components with Data-Control Flow Dependencies

To get a tight and safe WCET estimation, we need to analyze the control flow of the system in more detail. In this step, we resolve intra/inter-component data flow - control flow dependencies, try to find bounds of cycles, identify mutually exclusive paths etc. As depicted in Figure 1, the execution time of component A depends on the values appearing at its input ports *RUN* and *imax* respectively. The execution time of B depends on the outputs of A and depends on *RUN*. One can imagine that the output of B is further used by other components (however this case is not shown in Figure 1). This extract is why the outputs of B are also worth investigating.

We use abstract interpretation [4,5] in order to find out (a) relationships between input and output contexts of components (b) and information about the worst case execution path. The path information yielded by the abstract interpretation are passed as functionality constraints to the ILP solver. Assuming for example that component A is used in a context where *RUN=true*, we derive that $d_3 = d_2$ and $d_{12} = 0$ (see Figure 1).

3.1 Fundamentals of Abstract Interpretation

Abstract interpretation [4,5] is a theory of sound approximation of the semantics of programs. It can be viewed as a partial execution of a program, gaining information about its semantics without performing all the calculations. Although this approach can, in theory, deliver perfect results, the number of possible paths grows exponentially for every conditional jump and every loop iteration. To avoid the exploding nnumber of paths that have to be considered the intermediate data is merged at certain program points (such as at the end of loops) to safely approximate the results.

The meaning of a language is given as functions for the statements of the language computing over a concrete domain. A domain is a partially ordered set of values. For such a (concrete) language semantics, we define an abstract version which consists of an abstract value domain and simpler abstract functions working on the abstract values. The concrete meaning of a subset of the language ST is described in Section 3.2; while Section 3.3 introduces an abstract version. For more details about abstract interpretation see [4,5].

3.2 Syntax and Semantics of a ST Language

In this section we describe the syntax and semantics of a subset of the ST language. Only those elements are considered here which are necessary to analyze our example scenario shown in Figure 1.

- **Variables, types and values:** In this paper we limit ourself to the case that variables are either boolean ($BOOL$) or integer typed (INT). By Var, we denote the set of variables. Var_{BOOL} contains all boolean typed variables and Var_{INT} contains all integer typed variables. ($Var = Var_{BOOL} \cup Var_{INT}$). A function σ describes the values of a set of variables Var. σ corresponds to the *State*.
  · $\sigma_{INT} : Var_{INT} \rightarrow \mathbb{Z}$
  · $\sigma_{BOOL} : Var_{BOOL} \rightarrow \mathbb{B}$
  · $\sigma = \sigma_{INT} \cup \sigma_{BOOL}$
- **Arithmetic expressions:** An arithmetic expression a is either an integer constant $z \in \mathbb{Z}$, or a variable $x \in Var$, or it consists of two arithmetic expressions a_1, a_2 bound by a binary operator *op*. In this paper we use only $+$ and $-$ operators. The arithmetic expressions are:
  · INT-constants
  · $x \in Varvariable$

· two arithmetical expressions (a_1, a_2) bound by a binary operator: $a_1 + a_2$, $a_1 - a_2$

We denote the set of all arithmetic expressions by $AExp$. A function $\mathcal{A} : AExp \times State \rightarrow \mathbb{Z}$ evaluates an arithmetic expression:

$$\mathcal{A}(a,\sigma) = \begin{cases} \mathcal{A}(a_1,\sigma) + \mathcal{A}(a_2,\sigma) & \text{if } (a = a_1 + a_2), \\ \mathcal{A}(a_1,\sigma) - \mathcal{A}(a_2,\sigma) & \text{if } (a = a_1 - a_2), \\ a & \text{if } a \in Var_{INT}, \\ const_{INT} & \text{if } const_{INT} \in \{min_{INT} \ldots max_{INT}\} \end{cases} .$$

For instance, if $\sigma(i) = 8$ and $\sigma(j) = 10$ then $\mathcal{A}(i + j * 8, \sigma) = 8 + 10 * 8 = 88$.

- **Boolean expressions:** A boolean expression is either (a) $\neg b, b_1 \vee b_2, b_1 \wedge b_2$, where b_1 and b_2 are boolean expressions; (b) or $a_1 > a_2, a_1 \geq a_2, a_1 < a_2, a_1 \leq a_2, a_1 = a_2$, where a_1, a_2 are arithmetic expressions; (c) or $true, false$. We denote the set of all boolean expressions by $BExp$. $\mathcal{B} : BExp \times State \rightarrow \{true, false\}$ evaluates boolean expressions.

$$\mathcal{B}(b,\sigma) = \begin{cases} \mathcal{B}(b_1,\sigma) \wedge \mathcal{B}(b_2,\sigma) & \text{if } (b = b_1 \wedge b_2), \\ \mathcal{B}(b_1,\sigma) \vee \mathcal{B}(b_2,\sigma) & \text{if } (b = b_1 \vee b_2), \\ \neg\mathcal{B}(b_1,\sigma) & \text{if } (b = \neg b_1), \\ \mathcal{A}(a_1,\sigma) > \mathcal{A}(a_2,\sigma) & \text{if } (b = a_1 > a_2), \\ \mathcal{A}(a_1,\sigma) \geq \mathcal{A}(a_2,\sigma) & \text{if } (b = a_1 \geq a_2), \\ \mathcal{A}(a_1,\sigma) < \mathcal{A}(a_2,\sigma) & \text{if } (b = a_1 < a_2), \\ \mathcal{A}(a_1,\sigma) \leq \mathcal{A}(a_2,\sigma) & \text{if } (b = a_1 \leq a_2), \\ \mathcal{A}(a_1,\sigma) = \mathcal{A}(a_2,\sigma) & \text{if } (b = a_1 = a_2), \\ b & \text{if } b \in Var_{BOOL}, \\ const_{BOOL} & \text{if } const_{BOOL} \in \{true, false\} \end{cases} .$$

- **Statements:** In this paper we limit to a subset of the available statements of the ST language. An ST-based component consists of:
  · **Assignment statement:** $x := a;$, where $x \in Var$ and $a \in AExp$
  · **Sequence:** S1; S2; where S1 and S2 are statement blocks.
  · **Conditional statement:** IF (b) THEN S1; ELSE S2; END_IF; where $b \in BExp$ and S1, S2 are ST-programs.
- **Operational semantics:** We use small-step-semantics description, which simulates the execution step-by-step. $< S, \sigma > \rightarrow < S', \sigma' >$ denotes a transition from a state σ of a program to state σ' during step $S \rightarrow S'$. Termination of a program

is denoted by $< S, \sigma > \rightarrow < \sigma' >$. The rules of the semantics are given by:

$$< x := a, \sigma > \rightarrow \sigma[x \mapsto \mathcal{A}(a, \sigma)] \quad (ass)$$

$$< exit, \sigma > \rightarrow \sigma \quad (exit)$$

$$\frac{< S1, \sigma > \rightarrow < S1', \sigma' >}{< S1; S2, \sigma > \rightarrow < S1'; S2, \sigma' >} \quad (seq1)$$

$$\frac{< S1, \sigma > \rightarrow < \sigma' >}{< S1; S2, \sigma > \rightarrow < S2, \sigma' >} \quad (seq2)$$

$$< \text{IF (b) THEN S1 ELSE S2 END\_IF}, \sigma > \rightarrow < S1, \sigma >, if \mathcal{B}(b, \sigma) = true \quad (if1)$$

$$< \text{IF (b) THEN S1 ELSE S2 END\_IF}, \sigma > \rightarrow < S2, \sigma >, if \mathcal{B}(b, \sigma) = false \quad (if1)$$

- **Transition function:** if S and S' are two ST-programs and $\Sigma \subseteq State$, then $next$ is a transition function:

$$next_{S,S'}(\Sigma) = \{\sigma' | \sigma \in \Sigma \wedge < S, \sigma > \rightarrow < S', \sigma' >\}$$

$$next_{S,}(\Sigma) = \{\sigma' | \sigma \in \Sigma \wedge < S, \sigma > \rightarrow < \sigma' >\}$$

3.3 Abstract Semantics of ST Language

- **Abstract values:** We approximate the set of integers during the abstract interpretation by $[l, u]$ intervals, where l is the lower bound and u is the upper bound of the interval and $min_{INT} \leq l \leq u \leq max_{INT}$. *AbsState* corresponds to the abstract state of the program.

$$\begin{aligned} Interval =& \{[l, u], l, u \in \mathbb{Z} \wedge min_{INT} \leq l \leq u \leq max_{INT}\} \\ & \cup \{[l, \infty], l \in \mathbb{Z} \wedge min_{INT} \leq l \leq max_{INT}\} \\ & \cup \{[-\infty, u], u \in \mathbb{Z} \wedge min_{INT} \leq u \leq max_{INT}\} \\ & \cup \bot \qquad (\bot \text{ denotes an empty interval}) \\ & \cup \top \qquad (\top = [-\infty, \infty]) \end{aligned}$$

We define abstract elementary operations on the abstract values. In this paper we use only $\widetilde{+}$ and $\widetilde{-}$ ($\widetilde{*}$ and $\widetilde{/}$ are not considered). We define them as follows:

$$[l_1, u_1] \widetilde{+} [l_2, u_2] = [l_1 + u_1, l_2 + u_2]$$

$$[l_1, u_1] \widetilde{-} [l_2, u_2] = [l_1 - u_1, l_2 - u_2]$$

- **Evaluation of abstract arithmetic expressions:** Abstract arithmetic expressions are evaluated in the following way:

$$\widetilde{\mathcal{A}}(a, \rho) = \begin{cases} \widetilde{\mathcal{A}}(a_1, \rho) \widetilde{+} \widetilde{\mathcal{A}}(a_2, \rho) & \text{if } (a = a_1 + a_2), \\ \widetilde{\mathcal{A}}(a_1, \rho) \widetilde{-} \widetilde{\mathcal{A}}(a_2, \rho) & \text{if } (a = a_1 - a_2), \\ (a, \rho) & \text{if } a \in Var_{INT}, \\ [a, a] & \text{if } a \in \mathbb{Z}, \\ \bot & \text{if } \rho = \bot_\sigma \end{cases} .$$

- **Abstract transitions:** Similar to the concrete semantics, we define the abstract counterpart. Note, that in if_1 and if_2, the abstract values were deleted that would have triggered the execution of the other branch.

$$< [x := a], abs > \Rightarrow < \{\rho[x \mapsto \widetilde{a}] | \rho \in abs, \widetilde{a} \in \widetilde{\mathcal{A}}(a,\rho)\} > \quad (assignment)$$

$$\frac{< S1, abs > \Rightarrow < S1', abs' >}{< S1; S2, abs > \Rightarrow < S1'; S2, abs' >} \quad (seq_1)$$

$$\frac{< S1, abs > \Rightarrow < abs' >}{< S1; S2, abs > \Rightarrow < S2, abs' >} \quad (seq_2)$$

$$< exit, abs > \Rightarrow < abs > \quad (exit)$$

$$< \text{IF b THEN S1 ELSE S2 END\_IF}, abs > \Rightarrow < S1, abs \setminus \{\rho | \widetilde{\mathcal{B}}(b,\rho) = 0\} > \quad (if_1)$$

$$< \text{IF b THEN S1 ELSE S2 END\_IF}, abs > \Rightarrow < S2, abs \setminus \{\rho | \widetilde{\mathcal{B}}(b,\rho) = 1\} > \quad (if_2)$$

Based on this we can define the abstract $\widetilde{next}$:

$$\widetilde{next}_{S,S'}(abs) = \begin{cases} abs' & \text{if } < S, abs > \Rightarrow < S', abs' > \\ \emptyset & \text{else} \end{cases}.$$

$$\widetilde{next}_{S,}(abs) = \begin{cases} abs' & \text{if } < S, abs > \Rightarrow < abs' > \\ \emptyset & \text{else} \end{cases}.$$

3.4 Use of Abstract Interpretation

In the following we show, how the example scenario in Figure 1 can be analyzed using abstract interpretation. Before we do that, we instrument the CFG of the components with counter variables. The $x_i := x_i + 1$ lines injected at several program points on the CFG shown in Figure 4. Assuming that such a program would run on a PLC, the counter variables would - after termination of the program - hold the number of times that the corresponding path was executed during the run. If the program is analyzed using abstract interpretation, these variables will hold approximated values in the form of intervals. The idea using counter variables during abstract interpretation was taken from [9].

Table 1 lists an example run of component A with inputs $imax = [4,6]$ and $RUN = [true, false]$. Each column shows the history of variable values during the steps of the abstract interpretation. Values in a row correspond to the state before the step is executed. Note that abstract values are merged at certain program points such as at joining paths and at the loop headers. See [4,5] for more details about merging. After termination, the outputs of component A show the approximated values: $sum = [0,21]$: sum is something between [0, 21]; run and $imax$ were not changed during execution; $x1 = [1,1]$: the components entry point is executed once, $x2 = [0,1]$: we can not say for sure whether the THEN-branch of the IF condition is executed since the input run is either $true$ or $false$; $x3 = [0,6]$: the body of the loop is executed 6 times at most. Using this information we can derive functionality constraints for the IPET analysis introduced in Section 2.

step	program point	sum	run	i	imax	x1	x2	x3
1	sum := 0;..	[0, 0]	[true, false]	[0, 0]	[4, 6]	[0, 0]	[0, 0]	[0, 0]
2	x1 := x1 + 1;..	[0, 0]	[true, false]	[0, 0]	[4, 6]	[0, 0]	[0, 0]	[0, 0]
3	IF (RUN);..	[0, 0]	[true, false]	[0, 0]	[4, 6]	[1, 1]	[0, 0]	[0, 0]
4	x2 := x2 + 1;..	[0, 0]	[true]	[0, 0]	[4, 6]	[1, 1]	[0, 0]	[0, 0]
5	i := 1;..	[0, 0]	[true]	[0, 0]	[4, 6]	[1, 1]	[1, 1]	[0, 0]
7	IF (i < imax);..	[0, 0]	[true]	[1, 1]	[4, 6]	[1, 1]	[1, 1]	[0, 0]
8	x3 := x3 + 1;..	[0, 0]	[true]	[1, 1]	[4, 6]	[1, 1]	[1, 1]	[0, 0]
9	sum := sum + i;..	[0, 0]	[true]	[1, 1]	[4, 6]	[1, 1]	[1, 1]	[1, 1]
10	i := i + 1;..	[1, 1]	[true]	[1, 1]	[4, 6]	[1, 1]	[1, 1]	[1, 1]
11	IF (i ≤ imax);..	[0, 1]	[true]	[1, 2]	[4, 6]	[1, 1]	[1, 1]	[0, 1]
12	x3 := x3 + 1;..	[0, 1]	[true]	[1, 2]	[4, 6]	[1, 1]	[1, 1]	[0, 1]
13	sum := sum + i;..	[0, 1]	[true]	[1, 2]	[4, 6]	[1, 1]	[1, 1]	[1, 2]
14	i := i + 1;..	[1, 3]	[true]	[1, 2]	[4, 6]	[1, 1]	[1, 1]	[1, 2]
15	IF (i ≤ imax);..	[0, 3]	[true]	[1, 3]	[4, 6]	[1, 1]	[1, 1]	[0, 2]
16	x3 := x3 + 1;..	[0, 3]	[true]	[1, 3]	[4, 6]	[1, 1]	[1, 1]	[0, 2]
17	sum := sum + i;..	[0, 3]	[true]	[1, 3]	[4, 6]	[1, 1]	[1, 1]	[1, 3]
18	i := i + 1;..	[1, 6]	[true]	[1, 3]	[4, 6]	[1, 1]	[1, 1]	[1, 3]
19	IF (i ≤ imax);..	[0, 6]	[true]	[1, 4]	[4, 6]	[1, 1]	[1, 1]	[0, 3]
20	x3 := x3 + 1;..	[0, 6]	[true]	[1, 4]	[4, 6]	[1, 1]	[1, 1]	[0, 3]
21	sum := sum + i;..	[0, 6]	[true]	[1, 4]	[4, 6]	[1, 1]	[1, 1]	[1, 4]
22	i := i + 1;..	[1, 10]	[true]	[1, 4]	[4, 6]	[1, 1]	[1, 1]	[1, 4]
23	IF (i ≤ imax);..	[0, 10]	[true]	[1, 5]	[4, 6]	[1, 1]	[1, 1]	[0, 4]
24	x3 := x3 + 1;..	[0, 10]	[true]	[1, 5]	[4, 6]	[1, 1]	[1, 1]	[0, 4]
25	sum := sum + i;..	[0, 10]	[true]	[1, 5]	[4, 6]	[1, 1]	[1, 1]	[1, 5]
26	i := i + 1;..	[1, 15]	[true]	[1, 5]	[4, 6]	[1, 1]	[1, 1]	[1, 5]
27	IF (i ≤ imax);..	[0, 15]	[true]	[1, 6]	[4, 6]	[1, 1]	[1, 1]	[0, 5]
28	x3 := x3 + 1;..	[0, 15]	[true]	[1, 6]	[4, 6]	[1, 1]	[1, 1]	[0, 5]
29	sum := sum + i;..	[0, 15]	[true]	[1, 6]	[4, 6]	[1, 1]	[1, 1]	[1, 6]
30	i := i + 1;..	[1, 21]	[true]	[1, 6]	[4, 6]	[1, 1]	[1, 1]	[1, 6]
31	IF (i ≤ imax);..	[0, 21]	[true]	[1, 7]	[4, 6]	[1, 1]	[1, 1]	[0, 6]
32	end	[0, 21]	[true, false]	[0, 7]	[4, 6]	[1, 1]	[0, 1]	[0, 6]

Table 1
Abstract interpretation of component A

Now we can analyze component B (See Table 2). We use the approximated output value $sum = [0, 21]$ as input data context for component B. The analysis is quite straightforward since the control flow of B does not contain any loops. The results can be interpreted similarly as before: sum, run remain unchanged; $x4 = [1, 1]$: the entry point of component B is executed with certainty; $x6 = [0, 0]$ says that the THEN path of conditional $IF sum > 200$ is never executed.

We see in the first example, that it took quite long (32 steps) to approximate the variable/counter values of components A and B. We can speed up the approximation sequence using widening and narrowing operators as described in [5,4]. Table 3 and Table 4 show a faster approximation sequence started with the same component inputs as previously. (Note that we use here split intervals instead of ordinary ones.) Although we have a faster approximation sequence, the path information is more inaccurate (see for example the $sum = [0, \infty]$-output of component A). Note, that the abstract interpretation process also recognized, that the output

step	program point	sum	run	out	x4	x5	x6	x7
1	x4 := x4 + 1;..	[0, 21]	[true, false]	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
2	IF (RUN);..	[0, 21]	[true, false]	[0, 0]	[1, 1]	[0, 0]	[0, 0]	[0, 0]
3	x5 := x5 + 1;..	[0, 21]	[true]	[0, 0]	[1, 1]	[0, 0]	[0, 0]	[0, 0]
4	IF (SUM > 200)	[0, 21]	[true]	[0, 0]	[1, 1]	[1, 1]	[0, 0]	[0, 0]
5	x7 := x7 + 1;..	[0, 21]	[true]	[0, 0]	[1, 1]	[1, 1]	[0, 0]	[0, 0]
6	out := sum;..	[0, 21]	[true]	[0, 0]	[1, 1]	[1, 1]	[0, 0]	[1, 1]
7	end	[0, 21]	[true, false]	[0, 21]	[1, 1]	[0, 1]	[0, 0]	[0, 1]

Table 2
Abstract interpretation of component B started with outputs of component A

step	program point	sum	run	i	imax	out	x1	x2	x3
1	sum := 0;	[0, 0]	[true, false]	[0, 0]	[4, 6]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
2	x1 := x1 + 1;	[0, 0]	[true, false]	[0, 0]	[4, 6]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
3	IF (RUN);	[0, 0]	[true, false]	[0, 0]	[4, 6]	[0, 0]	[1, 1]	[0, 0]	[0, 0]
4	x2 := x2 + 1;	[0, 0]	[true]	[0, 0]	[4, 6]	[0, 0]	[1, 1]	[0, 0]	[0, 0]
5	i := 1;	[0, 0]	[true]	[0, 0]	[4, 6]	[0, 0]	[1, 1]	[1, 1]	[0, 0]
6	IF (i $\leq$ imax);	[0, ∞]	[true]	[1, ∞]	[4, 6]	[0, 0]	[1, 1]	[1, 1]	[0, ∞]
7	x3 := x3 + 1;	[0, ∞]	[true]	[1, 6]	[4, 6]	[0, 0]	[1, 1]	[1, 1]	[0, ∞]
8	sum := sum + i;	[0, ∞]	[true]	[1, 6]	[4, 6]	[0, 0]	[1, 1]	[1, 1]	[1, ∞]
9	i := i + 1;	[1, ∞]	[true]	[1, 6]	[4, 6]	[0, 0]	[1, 1]	[1, 1]	[1, ∞]
10	IF (i *leq* imax);	[0, ∞]	[true]	[1, 7]	[4, 6]	[0, 0]	[1, 1]	[1, 1]	[0, ∞]
11	end	[0, ∞]	[true, false]	[0, 0] U [7, 7]	[4, 6]	[0, 0]	[1, 1]	[0, 1]	[0, ∞]

Table 3
Abstract interpretation of component A with faster approximation sequence

step	program point	sum	run	out	x4	x5	x6	x7
1	x4 := x4 + 1;	[0, ∞]	[true, false]	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
2	IF (RUN)	[0, ∞]	[true, false]	[0, 0]	[1, 1]	[0, 0]	[0, 0]	[0, 0]
3	x5 := x5 + 1;	[0, ∞]	[true]	[0, 0]	[1, 1]	[0, 0]	[0, 0]	[0, 0]
4	IF (sum > 200)	[0, ∞]	[true]	[0, 0]	[1, 1]	[1, 1]	[0, 0]	[0, 0]
5	x6 := x6 + 1;	[201, ∞]	[true]	[0, 0]	[1, 1]	[1, 1]	[0, 0]	[0, 0]
6	out := 0;	[201, ∞]	[true]	[0, 0]	[1, 1]	[1, 1]	[1, 1]	[0, 0]
7	x7 := x7 + 1;	[0, 200]	[true]	[0, 0]	[1, 1]	[1, 1]	[0, 0]	[0, 0]
8	out := sum;	[0, 200]	[true]	[0, 0]	[1, 1]	[1, 1]	[0, 0]	[1, 1]
9	end	[0, ∞]	[true, false]	[0, 200]	[1, 1]	[0, 1]	[0, 1]	[0, 1]

Table 4
Abstract interpretation of component B started with outputs of component A

out of component *B* is certainly between 0 and 200.

4 Related Work

Related work can be grouped into two families:

(a) There are several approaches used to approximate the WCET of (real-time) software. Generally spoken usual WCET approaches combine a control flow analysis with a low level analysis. Control flow analysis is necessary to find the worst case scenario of the system. Several approaches are known, ranging from different kinds of user defined path annotations [1,7,22] to automatic analysis frameworks

[13,13,9,19,8]. The purpose of low level analysis is to determine the timing behavior of instructions assuming that the architectural features of the target system are known. For modern CPUs, it is important to deal with effects of various features like pipelines [35,34,14], caches [38,23,37,3] and branch predictors [25,33]. There are special kinds of approaches using measurements [28,18,30] or predicting probability based WCET instead of a single tight WCET value [29,2].

(b) The second family of approaches deal with prediction of various non-functional properties of component based systems. [31] develop a parametric contracts based approach for component systems. Several approaches built upon this model and predict properties such as reliability [32] and mean service execution time [12]. [15] deal similar to us with IEC61131-systems, however they neither investigate complex control flow in components nor data flow - control flow dependencies between them. In our approach, we also look at the inside structure of components to find out data flow - control flow dependencies between them. We believe that *tight* WCET approximation can be done if as many features of PLC systems as possible are considered.

5 Conclusion and Future Work

We showed in this paper how data flow - control flow dependencies between components can be analyzed using abstract interpretation. The approach can be also applied to some extent to the analysis of complex control flow inside of components. We showed the application of our approach on a practical example taken from the PLC domain. In our future work we extend our abstract interpretation based approach to get a tighter prediction. Furthermore that we also plan to develop an approach which allows to derive data-context dependent execution time information in a highly automated fashion. The main benefit of such an approach would be that this information could be reused during the prediction of the WCET property of the system, without analyzing the components structure again.

References

[1] J. Gustaffson A. Ermedahl. Deriving annotations for tight calculation of time. In *3rd International European Conference on Parallel Processing*, pages 1298–1307. LNCS 1300, 1997.

[2] G. Bernat, A. Colin, and SM Petters. pWCET: a tool for probabilistic worst case execution time analysis of real–time systems. In *Proceedings of the 3rd Intl. Workshop on Worst Case Execution Time Analysis. Porto, Portugal*, 2003.

[3] M. Campoy, A.P. Ivars, and JV Busquets-Mataix. Static use of locking caches in multitask preemptive real-time systems. In *Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, 2001.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[5] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978.

[6] C Ferdinand, R Heckmann, M Langenbach, and F Martin. Efficient analysis of pipeline models for wcet computation. In *5th International Workshop on Worst-Case Execution Time Analysis*, Palma de Mallorca, Spain, 2005.

[7] C. Ferdinand, R. Heckmann, and H. Theiling. Convenient User Annotations for a WCET Tool. 2003.

[8] Gustafsson, Lisper, Sandberg, and Bermudo. A tool for automatic flow analysis of c-programs for wcet calculation. In *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 106–112, 1 2003.

[9] J. Gustafsson, N. Bermudo, and L. Sjoberg. Flow Analysis for WCET calculation. Technical report, Technical Report 0547, ASTEC Competence Center, Uppsala University, URL: http://www. mrtc. mdh. se/publications/0547. ps, March 2003.

[10] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs, 1998.

[11] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation.* PhD thesis, May 2000.

[12] J. Happe. Predicting Mean Service Execution Times of Software Components Based on Markov Models. *Lecture Notes in Computer Sciense*, 3712:53, 2005.

[13] C. Healy, M. Sjodin, V. Rustagi, and D. Whalley. Bounding Loop Iterations for Timing Analysis. In *Proc. 4th Real-Time Technology and Applications Symp*, pages 12–21, 1998.

[14] CA Healy, RD Arnold, F. Mueller, DB Whalley, and MG Harmon. Bounding pipeline and instruction cache performance. *Computers, IEEE Transactions on*, 48(1):53–70, 1999.

[15] Jue Xie Heinz W. Schmidt, Ian D. Peake. Modelling predictable component-based distributed control architectures. In *Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS' 03), IEEE*, pages 339–346, 2004.

[16] Ralf Reussner Heinz W. Schmidt, Bernd J. Krmer. Predictable component architectures using dependent finite state machines, rissef 2002, lncs 2941. In *Proceedings of the NATO Workshop Monterey 2002, Radical Innovations of Software and Systems Engineering in the Future*, pages 362–378, Universita Ca Foscari di Venezia, Dipartimento Informatica, 2002.

[17] IEC. Programmable controllers - part 3: Programming languages, 2003.

[18] I. Puaut J-F. Deverge. Safe measurement-based wcet estimation. *5th International Workshop on Worst Case Execution Time Analysis*, 2005.

[19] B. Lisper J. Gustaffson. A prototype tool for flow analysis of c programs. *2nd International Workshop on Worst-Case Execution Time Analysis*, 2002.

[20] Viktoria Firus Jens Happe. Using stochastic petri nets to predict quality of service attributes of component-based software architectures. In *Tenth International Workshop on Component Oriented Programming, WCOP2005*, 2005.

[21] Michael Tiegelkamp Karl-Heinz John. *IEC 61131-3: programming industrial automation systems.* Berlin ; New York : Springer, 3 edition, 2001.

[22] R. Kirner and P. Puschner. Classification of Code Annotations and Discussion of Compiler-Support for Worst-Case Execution Time Analysis. 2005.

[23] M. Langenbach, C. Ferdinand, and R. Wilhelm. Worst case Execution Time Prediction. In *Proc. 2 ndInternational Workshop on Worst-Case Execution Time Analysis,(WCET2002)*, 2002.

[24] Y.T.S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(3):257–279, 1999.

[25] T. Mitra and A. Roychoudhury. A Framework to Model Branch Prediction for WCET Analysis. In *2nd Workshop on Worst Case Execution Time Analysis (WCET)*, 2002.

[26] T. Nolte, A. Möller, and M. Nolin. Using components to facilitate stochastic schedulability analysis. In *Proceedings of the WiP Session of the 24th IEEE Real-Time System Symposium, Cancun, Mexico*, December 2003.

[27] S. V. Cavalcante P. A. Guedes. On the design of an extensible platform for flow analysis of java using abstract interpretation. In *3rd International Workshop on Worst Case Execution Time Analysis*, Portugal, 2003.

[28] S. Petters. Comparison of trace generation methods for measurement based wcet-analysis. *3rd International Workshop on Worst Case Execution Time Analysis*, 2003.

[29] S.M. Petters. How much Worst Case is Needed in WCET Estimation? In *2nd International Workshop on Worst Case Execution Time Analysis, Vienna, Austria, June*, 2002.

[30] P. Puschner R. Kirner. Measurement-based worst-case execution time analysis using automatic test-data generation. *4th International Workshop on Worst-Case Execution Time Analysis*, 2004.

[31] R.H. Reussner, V. Firus, and S. Becker. Parametric Performance Contracts for Software Components and their Compositionality. *Proceedings of the 9th International Workshop on Component-Oriented Programming (WCOP 04).(2004)*, 2004.

[32] R.H. Reussner, H.W. Schmidt, and I.H. Poernomo. Reliability prediction for component-based software architectures. *The Journal of Systems & Software*, 66(3):241–252, 2003.

[33] C. Rochange and P. Sainrat. Difficulties in computing the WCET for processors with speculative execution. 2002.

[34] J. Schneider and C. Ferdinand. Pipeline Behavior Prediction for Superscalar Processors. Technical report, Technical Report A/02/99, Uni. d. Saarlandes, February 1999.

[35] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 35–44, 1999.

[36] A. Tesanovic, D. Nystrom, J. Hansson, and C. Norstrom. Aspect-level wcet analyzer: a tool for automated wcet analysis of a real-time software composed using aspects and components, 2003.

[37] H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecturemodelling and program path analysis. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 144–153, 1998.

[38] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Real-Time Systems*, 18(2):157–179, 2000.