Principles for Value Annotation Languages

Björn Lisper

School of Innovation, Design and Engineering, Mälardalen University
Box 883, S-721 23 Västerås, Sweden.
bjorn.lisper@mdh.se

Abstract

Tools for code-level program analysis need formats to express various properties, like relevant properties of the environment where the analysed code will execute, and the analysis results. Different WCET analysis tools typically use tool-specific annotation languages for this purpose. These languages are often geared towards expressing properties that the particular tool can handle rather than being general, and mostly their semantics is only specified informally. This makes it harder for tools to communicate, as well as for users to provide relevant information to them. Here, we propose a small but general assertion language for *value constraints* including IPET flow facts, which is an important class of annotations for WCET analysis tools. We show how to express interesting properties in this language, we propose some syntactic conveniences, and we give the language a formal semantics. The language could be used directly as a tool-independent annotation language, or as a meta-language to give exact semantics to existing value annotation and flow fact formats.

1998 ACM Subject Classification C.3 Special-Purpose and Application-Based Systems

Keywords and phrases Real-Time System, WCET analysis, Flow Fact, Assertion

Digital Object Identifier 10.4230/OASIcs.WCET.2014.1

1 Introduction

WCET analysis tools provide means to estimate the WCET of code with increased confidence, safety and automation compared with a manual analysis. Alas, full automation is hard to attain due to a number of reasons. Some are fundamental, such as the undecidability of the WCET analysis problem, others are of more practical nature like the need to provide relevant information not present in the code, or give directives to fine-tune the analysis. Thus, WCET analysis tools have annotation languages to provide various kinds of information to the analysis. Examples are:

- annotations providing *information about the environment*, like hardware configuration, entry points of tasks, etc.,
- annotations *directing the analysis* (like selection of abstract domain, context-sensitivity, choice of internal representations, kinds of generated flow facts),
- directives how to present the analysis results,
- *value annotations* constraining the possible values of program variables in different program points, and
- *flow facts* constraining the possible program flows.

Unfortunately, the means to provide these kinds of information are not systematically developed. WCET analysis tools tend to have their own annotation languages, which may be apt to provide information for the respective tool but are not portable across tools. As for manually provided information, many of these tool-specific formats do not provide a particularly user-friendly syntax. The semantics is not always entirely clear either, due to the absence of formal definitions.

14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk; pp. 42–51
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

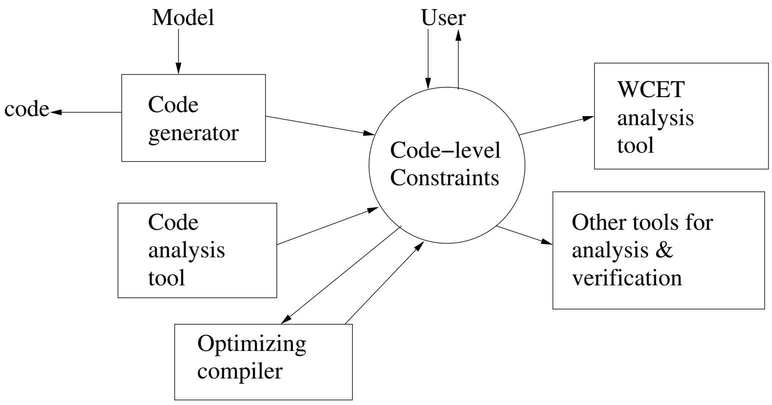

Figure 1 An ecosystem of embedded systems tools.

Furthermore, WCET analysis tools do not exist in isolation. Today one can speak of an "ecosystem" of code-level tools, as depicted in Fig. 1, including code generators for model-based development, general-purpose static code analysis tools, optimising compilers, and tools for formal verification. It is obvious that general formats for code-level constraints would be helpful for the integration of code-level tool chains.

The contribution of this paper is a step in this direction. We define a simple but general core language for *value constraints* from first principles. Such constraints provide information about reachable states, much like assertions in Floyd-Hoare logic [7, 8]. We take measures to make the constraint language as independent of the "host language" as possible, making it portable over a wide range of code formats. We make some suggestions for user-friendly syntax. The language can express numerical constraints on values of program variables, which is sufficient to express the value annotations and flow facts supported by current WCET tool annotation languages, but it can easily be extended to express constraints on other data types. We give the language a formal semantics, making minimal assumptions on the semantics of the host language. We also prove a theorem about compositionality of assertions.

2 Some Existing Annotation Languages

We now review the annotation languages for some existing WCET analysis tools.

FFX [3] is intended to be a portable WCET annotation language for flow facts. It is supported by TuBound [12], oRange [4], Otawa [1], and CalcWCET167 [10]. FFX represents the analyzed code as a structured XML document, where annotations appear as attributes in tags representing program constructs such as loops. FFX can specify upper loop bounds, and also whether or not they are exact. It is unclear whether it can represent more general linear flow facts, which describe relations between execution counters for different parts of the code. It can represent contexts: however, these seem not to be calling contexts but rather information about the environment such as the target hardware.

The annotation language of aiT [5], called AIS [6], can describe various kinds of flow facts, such as (upper) loop bounds, infeasible paths, and general linear flow constraints on IPET execution counters. Loop bounds can be complex expressions, and may for instance refer to execution counters of outer loops. Some context-sensitivity is provided by the ability to tie flow facts to certain call sites for functions. AIS can represent flow facts both for source code (C) and for executables: locations for execution counters are formed from file name and position in the file for source code, and are provided as numerical addresses for executables.

The WCET analysis tool SWEET [18] provides a rich set of different annotations. It can read value constraints in a certain format, constraining the values of program variables in certain program points to intervals. A common use is to specify value constraints on inputs. SWEET can also compute value constraints and export them using the same format. Furthermore SWEET provides a rich set of flow facts, including general linear constraints on execution counters. Execution counters can be local to certain execution contexts (typically loop or function bodies), and are then reset at each entry of the context. Flow facts involving such counters can be constrained to certain ranges of loop iterations. Flow facts involving global execution counters are also allowed. Flow facts can be constrained to be valid only for certain calling contexts, which are specified by explicit call strings: this provides context-sensitivity.

Bound-T [9] can use a number of different assertions. It can take value range constraints for program variables, bounds on the number of loop iterations, and function calls, “path not taken”-constraints, and different bounds on stack usage. Variables can also be asserted not to have their values changed in certain parts of the program. Assertions can be tied to program parts in an interesting way, identifying, e.g., loops by properties like that they use a certain variable, call a certain function, or whether a loop is the inner or outer loop in a loop nest. Assertions can be restricted to certain calling contexts and can thus be context-sensitive. Bounds on the number of executions can also be placed directly on instructions, using addresses or offsets.

These languages have some features in common. They can all express flow facts, of different generality. The flow facts may concern global IPET execution counters as well as local execution counters, for certain execution contexts. Some of them can also express certain kinds of value constraints. They provide varying levels of context-sensitivity. Our proposed core language aims to cover these aspects in a unified way.

3 A Wish List for a Language for Code-level State Constraints

Based on experience of WCET annotation languages as well as general language design, the following wish list on a value constraint language can be formulated:

- it should work over a wide range of code-level tools (not necessarily only for WCET analysis),
- it should work over a wide range of host languages, on different levels
- it should be general yet simple, extensible, and have few but powerful constructs,
- it should have a succinct, intuitive syntax for humans, as well as an easily machine-readable form (XML) for tools,
- it should be able to express general restrictions on flow facts, including the ability to express constraints in existing annotation languages by translation,
- it should be able to express different kinds of context sensitivity, and
- it should have a clear and simple formal semantics.

This wish list has guided the design of our core language.

4 A Starting Point: The Assertion Language of Floyd-Hoare Logic

A general, existing assertion language for constraints on program variable values is the one used in Floyd-Hoare logic [7, 8]. A good description is found in [19]. It has the following elements:

- program variables, which depend on program state,
- *auxiliary* variables, which are independent of state, and
- some sublanguage to define predicates over variables (typically consisting of boolean and arithmetic expressions, including quantifiers ($\forall$, $\exists$) over auxiliary variables, but whose elements may vary depending on what kind of assertions are to be expressed

An example of a statement in Floyd-Hoare logic is

$$\{X = i\}X := X + 1\{X = i + 1\}$$

This assertion is a triple of a *pre-condition*, a program, and a *post-condition*. X is a program variable and i an auxiliary variable. The statement expresses a relation that holds between preceding and succeeding states: for any value of i, if the value of X equals i in a state preceding the program, then it will equal $i + 1$ in the state that results after having executed the program.

Floyd-Hoare logic was originally defined for a simple, structured imperative language, and it comes with a set of inference rules, based on the syntax of the language, by which assertions can be proved deductively from sub-assertions of sub-programs. However the assertions can also be given a direct semantics in terms of state transitions, which is of interest here. See [19].

5 A Core Language for Value Constraints

Floyd-Hoare logic is defined over a high-level language where the control flow is decided entirely by the syntax. We want our core language for assertions to work over a wide variety of code formats, including low-level formats with unstructured control flow. Therefore we abstract away from the syntax of the host language, and we will make only minimal assumptions on its semantics. The abstract syntax for our assertion language is given by the following:

$$\begin{array}{lcl} a & ::= & n \mid i \mid X \mid a_1 \ aop \ a_2 \\ p & ::= & true \mid false \mid p_1 \wedge p_2 \mid p_1 \vee p_2 \mid \neg p \mid a_1 \ rop \ a_2 \mid \forall i.p \mid \exists i.p \mid PC = L \\ c & ::= & p_1 \rightarrow p_2 \end{array}$$

Here a stands for arithmetic expressions, and p predicates. n stands for numerical constants, i auxiliary variables, X program variables, aop arithmetic operators ($+$, $-$, $\ldots$), and rop relational operator ($<$, $=$, $\ldots$). L stands for *labels*, see below. We will freely use operators that can be derived from the core language, like implication ($\Longrightarrow$).

This part defines a predicate language over arithmetic expressions that is completely standard (except for the "$PC = L$" part, which will be explained below). Like in Floyd-Hoare logic the difference between program variables and auxiliary variables is that the value of a program variable depends on the state, whereas the values of auxiliary variables are independent of the state. We allow quantification over auxiliary variables, but not over program variables.

The statements of form $p_1 \rightarrow p_2$ are the assertions in our core language, and they correspond to the triples in Floyd-Hoare logic (with p_1 as pre- and p_2 as post-condition). The meaning of $p_1 \rightarrow p_2$ is "all states that are reachable from a state satisfying p_1 must satisfy p_2". We will provide an exact definition in Section 6.

We make the following assumptions on the host language and its semantics. It has program variables that can hold values, and there are labels that identify program points. At this point we assume nothing more about labels than that they can be compared for equality. The language has states: in each state σ, a program variable X holds a numerical value $\sigma(X)$. Furthermore there is a distinguished variable PC such that $\sigma(PC)$ is a label: thus, the state also contains the current position in the code. The semantics of a program in the host language is given by a set of state transitions $\sigma \to \sigma'$.

Our assertion language is defined over conditions on numerical expressions, but is easily extended to conditions over other data types. It should also be straightforward to give the language both a user-friendly syntax as well as a conveniently machine-readable XML format. We propose some syntactic conveniences in Section 5.2.

The language so far can express sensitivity to contexts that are conditions on the state (trough implication), but it does not have any means to express call-string contexts. We show how this can be added in Section 7.

Let us now see some examples of assertions. To make the examples more concrete we assume labels "*entry*", "*exit*" representing the entry and exit point of the host program, respectively:

- $(PC = \mathit{entry}) \to (PC = L \implies X < 17)$: for all states reachable from the start of the program, if at label L then $X < 17$;
- $(PC = \mathit{entry}) \to (PC = L \wedge 3 \leq I \leq 7 \implies X < 17)$: for all states reachable from the start of the program, if at label L, with the value of I between 3 and 7, then $X < 17$;
- $(PC = \mathit{entry} \wedge 1 \leq X \leq 10) \to (PC = \mathit{exit} \implies Y \leq 100)$: if the program is started with $1 \leq X \leq 10$ then, at exit, $Y \leq 100$;
- $(PC = L \wedge X = i) \to (PC = L' \implies X = 2 \cdot i)$: for any value of i, if the program passes L with $X = i$ then afterwards, whenever at L', $X = 2 \cdot i$;
- $\mathit{true} \to X < 32767$: a global invariant, in all states holds that $X < 32767$.

Notice how restrictions on inputs, like confining an input value to a certain range, can be expressed as arithmetic contraints in the condition defining the initial states. Also note how contexts that are restrictions to certain states can be expressed simply as antecedents in implications. Such restrictions can for instance be presence at a certain program point, or that the value of a loop counter is in a certain interval.

5.1 Labels

Labels can be basically anything that identifies program points. For high-level languages like C, labels can be explicit C labels defined in the source code, or they can be formed from file name, line number, and position on the line as in AIS. Another possibility is to use paths through the parse tree of the program as labels. For low-level code a label can be a pair (e, n) where e is a symbolic entry point and n is a numerical offset, or even a fully numerical address for a linked executable.

Our basic core language only assumes that labels can be compared for equality. Certain kinds of labels, like for instance numerical addresses, can allow a richer set of conditions to specify sets of labels.

Different kinds of labels can be *fragile* to different extent, in that they may be destroyed by recompilation or editing of the source code. Examples of fragile labels are numerical addresses in executables, and line numbers in source code. Assertions that use such labels may have to be restored frequently. While interesting, the construction of non-fragile labels is outside the scope of this paper.

5.2 Syntactic Sugar

The notation developed so far can be simplified for some common cases. For instance, it can be expected that restrictions to certain program points are frequent. Thus, using the notation "$@L$" for "$PC = L$" may help. It also seems like a common case to consider all the states that are reachable from the entry point of the program: thus on the top level, where an assertion is expected, one may allow to write p as a shorthand for $@entry \to p$. Some of our examples above can then be written:

- $@L \implies X < 17$ (understood, for all states reachable from the entry point)
- $@L \wedge 3 \le I \le 7 \implies X < 17$ (similarly)
- $(@entry \wedge 1 \le X \le 10) \to (@exit \implies Y \le 100)$
- $(@L \wedge X = i) \to (@L' \implies X = 2 \cdot i)$:

5.3 IPET Execution Counters and Flow Facts

IPET execution counters are "virtual" program variables that keep track of how many times a program part has been executed. Flow facts are expressed as arithmetic value constraints on these counters. The counters are typically defined relative to some execution context, with some entry and exit points, such that they are reset each time the execution context is entered and incremented by one each time the program part in question is executed. Global execution counters are defined relative to the whole program, with the *entry* label as entry point and *exit* as the exit point. For the final IPET calculation of the WCET estimate it is the possible values of the counters at exit that are of interest.

To express IPET execution counters we introduce the unary operator "#" on labels: if L is a label, then $\#L$ is the IPET counter associated with the program point of that label. We leave open how to associate IPET counters with different execution contexts, and how to exactly specify their semantics: for now we assume that they are global, and that their semantics is given by the informal description above.

We can now express flow facts in our assertion language, as value contraints on the IPET counters. Here are some examples:

- $@exit \implies \#L < 100$: a simple capacity constraint;
- $@exit \implies \#L = 99$: an exact capacity constraint;
- $@exit \implies \#L_1 + \#L_2 \le 1$: a mutual exclusivity constraint;
- $(@entry \wedge 1 \le X \le 10) \to (@exit \implies \#L \le 100)$: a capacity constraint under the condition that the value of X lies in the range $[1 \ldots 10]$ at entry;
- $(@entry \wedge X = n) \to (@exit \implies \#L \le 2 \cdot n + 1)$: a parametric capacity constraint relating the number of executions of L to the value of X at entry;
- $@exit\_local \wedge 3 \le \#L \le 7 \implies \#L_{local} < 17$: for each of the iterations 3 to 7 of an outer execution context with label L, L_{local} is executed less than 17 times.

In the last example $\#L_{local}$ is supposed to be a local execution counter, which is reset each time its local execution context is entered.

As the values of IPET counters at exit from their execution contexts are of primary interest, a possible syntactic simplification is to allow the "$@exit \implies$" part to be implicit and add it automatically when parsing a constraint that contains an IPET counter. So. for instance, the second constraint above could then simply be written $\#L = 99$, which then is to be interpreted as $@exit \implies \#L = 99$, which in turn stands for $@entry \to (@exit \implies \#L = 99)$.

5.4 Time

The state could also contain time. This gives the ability to express fine-grained real-time constraints on certain pieces of code. For instance L and L' may be program points in a loop, with loop counter variable I, such that we want to specify that within each iteration L' should never be executed more than 7 time units later than L. If time is represented by the program variable T, then this constraint can be expressed as

$$(@L \wedge t = T \wedge i = I) \rightarrow (@L' \wedge i = I \implies T - t \leq 7)$$

This example makes heavy use of auxiliary variables to refer to the value of a program variable in a pre-condition from the post-condition. This is quite common. A possible syntactic convenience is to make the equalities in the pre-condition implicit, and refer to the "old" value of program variable X as $X.old$ in the post-condition. With this notation, our example becomes

$$@L \rightarrow (@L' \wedge I = I.old \implies T - T.old \leq 7)$$

6 Formal Semantics

We now give a formal semantics to our core assertion language defined in Section 5. As is standard in programming language theory, we use semantic functions. These take three arguments: a syntactic form, an *interpretation* I that maps auxiliary variables to values, and a program state σ mapping program variables to values. The definitions of the semantic function $\mathcal{A}[\![\ ]\!]$, for arithmetic expressions, and $\mathcal{B}[\![\ ]\!]$, for boolean expressions (predicates), are completely standard and are given below for completeness (cf. [19]):

$$\begin{gathered}
\mathcal{A}[\![n]\!]\, I\, \sigma = n \quad \mathcal{A}[\![i]\!]\, I\, \sigma = I(i) \quad \mathcal{A}[\![X]\!]\, I\, \sigma = \sigma(X) \\
\mathcal{A}[\![a_1\ aop\ a_2]\!]\, I\, \sigma = \mathcal{A}[\![a_1]\!]\, I\, \sigma\ aop\ \mathcal{A}[\![a_2]\!]\, I\, \sigma \\
\mathcal{B}[\![true]\!]\, I\, \sigma = true \quad \mathcal{B}[\![false]\!]\, I\, \sigma = false \quad \mathcal{B}[\![p_1 \wedge p_2]\!]\, I\, \sigma = \mathcal{B}[\![p_1]\!]\, I\, \sigma \wedge \mathcal{B}[\![p_2]\!]\, I\, \sigma \\
\mathcal{B}[\![p_1 \vee p_2]\!]\, I\, \sigma = \mathcal{B}[\![p_1]\!]\, I\, \sigma \vee \mathcal{B}[\![p_2]\!]\, I\, \sigma \quad \mathcal{B}[\![\neg p]\!]\, I\, \sigma = \neg \mathcal{B}[\![p]\!]\, I\, \sigma \\
\mathcal{B}[\![a_1\ rop\ a_2]\!]\, I\, \sigma = \mathcal{A}[\![a_1]\!]\, I\, \sigma\ rop\ \mathcal{A}[\![a_2]\!]\, I\, \sigma \quad \mathcal{B}[\![\forall i.p]\!]\, I\, \sigma = \forall n.(\mathcal{B}[\![p]\!]\, I[n/i]\, \sigma) \\
\mathcal{B}[\![\exists i.p]\!]\, I\, \sigma = \exists n.(\mathcal{B}[\![p]\!]\, I[n/i]\, \sigma) \quad \mathcal{B}[\![PC = L]\!]\, I\, \sigma = \sigma(PC) = L
\end{gathered}$$

Here, $I[n/i]$ stands for the interpretation that maps i to n but otherwise behaves like I.

We now give the semantic function $\mathcal{C}[\![\ ]\!]$ for assertions $p_1 \rightarrow p_2$. The definition uses the relation $\rightarrow^*$ on states, defined by $\sigma \rightarrow^* \sigma'$ if and only if σ' is reached from σ through zero or more state transitions (reflexive-transitive closure of the transition relation $\rightarrow$):

$$\mathcal{C}[\![p_1 \rightarrow p_2]\!] = \forall I, \sigma, \sigma'.[(\mathcal{B}[\![p_1]\!]\, I\, \sigma \wedge \sigma \rightarrow^* \sigma') \implies \mathcal{B}[\![p_2]\!]\, I\, \sigma'] \tag{1}$$

Thus $p_1 \rightarrow p_2$ if, for each state σ where p_1 holds, and for each state σ' that is reachable from σ, p_2 holds for σ'.

▶ **Theorem 1** (Compositionality). $p_1 \rightarrow p_2 \wedge p_2 \rightarrow p_3 \implies p_1 \rightarrow p_3$.

Proof. Assume that $p_1 \rightarrow p_2$ and $p_2 \rightarrow p_3$. $\sigma \rightarrow^* \sigma$ for all states σ. Thus, since $p_1 \rightarrow p_2$, p_2 holds for all states where p_1 holds. Since $p_2 \rightarrow p_3$ it follows that p_3 holds for each state reachable from a state where p_1 and thus also p_2 holds, thus it must hold that $p_1 \rightarrow p_3$. ◀

Theorem 1 implies that assertions for a program can be composed out of assertions on its parts, much like the inference rules for Floyd-Hoare logic.

7 Context Sensitivity

Value annotations and flow facts can be made more precise if context-sensitive. The contexts we have seen so far are sets of states defined by p' in an assertion $p \to (p' \implies p'')$, which loosens the requirement on p'' to hold merely for the reachable states where p' holds. However, another very important class of contexts are *calling contexts*. These can be used to express that a value annotation is to hold only when a function is called in a certain way, perhaps through a specific chain of other function calls. The well-known concept of *call-strings* [14] can be used to define such contexts.

In our setting a call-string is a sequence of special labels that identify particular program points like call sites for functions, or entry points to loops. Let $\mathbf{L}$ be the set of labels under consideration, and let $\mathbf{C} \subseteq \mathbf{L}$ be the set of labels that are considered to be call sites. A sequence $s \in \mathbf{C}$ is then a call-string. Let S be an expression that defines a set of call-strings $C(S) \subseteq \mathbf{C}$. The notation

$$p \to p' \textit{ through } S$$

is a suggested extension of the core language in Section 5 to denote an assertion where p' is to hold for those states, reachable from some state where p holds, through a sequence of states such that the sequence of traversed call sites belongs to $C(S)$. We will not be specific about the exact format of S: it could, for instance, be some kind of regular expression defining a set of call-strings.

For completeness we now give a formal semantics to this kind of assertion. In order to do this we need to develop some notation for different sequences. Given the transition relation "$\to$" on states, which describes the semantics of the host program, we define:

$$\begin{array}{rcl} \mathit{Paths}(\sigma,\sigma') & = & \{\, \sigma_1 \cdots \sigma_n \mid \sigma_1 = \sigma, \sigma_i \to \sigma_{i+1}, i = 1,\ldots,n-1, \sigma_n = \sigma' \,\} \\ \mathit{PC}(\sigma_1 \cdots \sigma_n) & = & \sigma_1(\mathit{PC}) \cdots \sigma_n(\mathit{PC}) \\ \mathit{Labels}(\sigma,\sigma') & = & \{\, \mathit{PC}(\sigma_1 \cdots \sigma_n) \mid \sigma_1 \cdots \sigma_n \in \mathit{Paths}(\sigma,\sigma') \,\} \end{array}$$

$\mathit{Paths}(\sigma,\sigma')$ is the set of sequences of states leading from σ to σ'. $\mathit{PC}(\sigma_1 \cdots \sigma_n)$ is the sequence of labels generated by the sequence of states $\sigma_1 \cdots \sigma_n$. $\mathit{Labels}(\sigma,\sigma')$ is the set of sequences of labels generated by the possible state transitions leading from σ to σ'. Finally we introduce the well-known projection operator "$\restriction$" on strings s and sub-alphabets A: $s \restriction A$ is the substring of s obtained from its characters in A appearing in the same order as in s. For instance, if $A = \{a,b,c\}$ then $adecbbe \restriction A = acbb$. We extend $\restriction$ to sets of strings S, viz.

$$S \restriction A = \{\, s \restriction A \mid s \in S \,\}$$

We can now extend (1) to call-string-sensitive assertions as defined above:

$$\mathcal{C}[\![p_1 \to p_2 \textit{ through } S]\!] = \forall I, \sigma, \sigma'.[(\mathcal{B}[\![p_1]\!]\, I\, \sigma \wedge (\mathit{Labels}(\sigma,\sigma') \restriction \mathbf{C}) \cap C(S) \neq \emptyset) \implies \mathcal{B}[\![p_2]\!]\, I\, \sigma']$$

Thus the assertion $p_1 \to p_2$ *through* S holds if, for all sequences of labels from a state σ where p_1 holds to another state σ', such that the projection of the sequence onto the set of call sites is a call-string defined by S, p_2 holds for σ'. Compared with (1), where p_2 is to hold for all states reachable from a state satisfying p_1, p_2 now only has to hold for states reachable through a call-string given by S.

Theorem 1 can be extended to the context-sensitive case. Define $S \cdot S'$ by $C(S \cdot S') = \{\, s \cdot s' \mid s \in C(S), s' \in C(S') \,\}$, where "$\cdot$" is concatenation of sequences: we then have the following result (proof straightforward, but omitted due to lack of space):

▶ **Theorem 2.** $p_1 \to p_2$ *through* $S \wedge p_2 \to p_3$ *through* $S' \implies p_1 \to p_3$ *through* $S \cdot S'$.

8 Related Work

We have already reviewed the annotation languages of some WCET analysis tools in Section 2. There are a number of others: a comprehensive review and classification of such languages is found in [11]. Most of them express flow facts as linear arithmetic constraints on execution counters, as here, varying from simple looop bounds to general linear constraints. A notable exception is the Information Description Language [15], which can specify sets of feasible paths by regular expressions. This makes it possible to specify the exact order of execution of different program parts, whereas constraints in IPET execution counters only constrain the number of times they can execute and not the exact order.

Assertions using pre- and post-conditions have been used for a long time in formal software development: a classical example is the Vienna Development Method [2]. Such assertions can also be seen as *contracts*: the pre-condition is then the assumption on the environment, and the post-condition is what the program guarantees if the assumption is fulfilled. Contracts are essential for reasoning about component-based software. The language Eiffel provides means to express contracts [13].

On model level, the language OCL is used to specify properties of UML models. A formal semantics is given in [16]. The specification language Z uses Zermelo's set theory to express properties of models and programs in a pre/post-condition style [17].

9 Conclusions and Further Research

We have presented a simple core language for expressing assertions in pre/post-condition style, making minimal assumptions on the host language. This language can express value constraints, and it can be extended with IPET execution counters yielding the capability to express flow facts. Parametric flow facts and value constraints can be expressed using the auxiliary variables of the assertion language. We also proposed a way to include constraints with call strings, making is possible to express context-sensitive assertions. Special care was taken to develop the formal semantics of the language, and we proved a theorem about compositionality of assertions. This theorem is of practical interest since it allows assertions for a program to be composed from assertions on its parts.

We strongly believe that there is a need for a simple and general code-level assertion language that is designed from first principles, and has a clear semantics. In the best of worlds, a standardised such language could be used to exchange code-level information between a variety of tools including WCET analysis tools. In any case it can help understanding the underlying principles of annotation languages, and be used to give them a precise semantics.

Acknowledgment

This work was partially supported by COST Action IC1202: Timing Analysis On Code-Level (TACLe), and by the Swedish Research Council project Contesse (2010-4276).

References

1 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *Proc. IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, pages 35–46. Springer, October 2010.

2 Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: the Meta-Language*, number 61 in Lecture Notes in Computer Science. Springer-Verlag, 1978.

3 Armelle Bonenfant, Hugues Cassé, Marianne de Michiel, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. FFX: A portable WCET annotation language. In *Proc. 20th International Conference on Real-Time and Network Systems (RTNS'12)*, pages 91–100, New York, NY, USA, 2012. ACM.

4 Marianne de Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *Proc. IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*, pages 161–168, Kaohsiung, Taiwan, August 2008. IEEE Computer Society.

5 Christian Ferdinand, Reinhold Heckmann, and Bärbel Franzen. Static memory and timing analysis of embedded systems code. In *Proc. 3^{rd} European Symposium on Verification and Validation of Software Systems (VVSS'07)*, number 07-04 in TUE Computer Science Reports, pages 153–163, Eindhoven, The Netherlands, March 2007.

6 Christian Ferdinand, Reinhold Heckmann, and Henrik Theiling. Convenient user annotations for a WCET tool. In Jan Gustafsson, editor, *Proc. 3^{rd} International Workshop on Worst-Case Execution Time Analysis (WCET'2003)*, Porto, July 2003.

7 R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Proc. Symp. Applied Mathematics*, vol. 19: *Mathematical Aspects of Computer Science*, pages 19–32, Providence, R.I., 1967. American Mathematical Society.

8 C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 583, October 1969.

9 Niklas Holsti and Sami Saarinen. Status of the Bound-T WCET tool. In *Proc. 2^{nd} International Workshop on Worst-Case Execution Time Analysis (WCET'2002)*, 2002.

10 Raimund Kirner. The WCET analysis tool CalcWcet167. In Tiziana Margaria and Bernhard Steffen, editors, *Proc. 5^{th} International Symposium on Leveraging Applications of Formal Methods (ISOLA'12)*, Lecture Notes in Computer Science, Heraclion, Crete, October 2012. Springer-Verlag.

11 Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. *Software & Systems Modeling*, 10(3):411–437, 2011.

12 Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. r-TuBound: Loop bounds for WCET analysis. In Nikolaj Bjørner and Andrei Voronkov, editors, *Proc. Logic for Programming, Artificial Intelligence, and Reasoning*, volume 7180 of *Lecture Notes in Computer Science*, pages 435–444. Springer, 2012.

13 Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

14 Flemming Nielson, Hanne Ries Nielson, and Chris Hankin. *Principles of Program Analysis, 2^{nd} edition.* Springer, 2005.

15 Chang Yun Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5(1):31–62, March 1993.

16 Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. In Tok-Wang Ling, Sudha Ram, and Mong Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, volume 1507 of *Lecture Notes in Computer Science*, pages 449–464. Springer-Verlag, 1998.

17 J Michael Spivey. *Understanding Z: a specification language and its formal semantics.* Number 3. Cambridge University Press, 1988.

18 SWEET home page, 2011. `www.mrtc.mdh.se/projects/wcet/sweet/`.

19 Glynn Winskel. *The Formal Semantics of Programming Languages – An Introduction.* MIT Press, 1993.