Certifying Execution Time

Vítor Rodrigues[2,3], João Pedro Pedroso[2], Mário Florido[2,3], Simão Melo de Sousa[1,3]

[1] RELiablE And SEcure Computation Group
Universidade da Beira Interior, Covilhã, Portugal
[2] DCC-Faculty of Science, Universidade do Porto, Portugal
[3] LIACC, Universidade do Porto, Portugal

Abstract. In this paper we present the framework *Abstraction-Carrying CodE Platform for Timing validation* (ACCEPT), designed for timing analysis of embedded real-time systems using the worst-case execution time (WCET) as the safety parameter. In the context of real-time embedded code safety, we describe in detail the component responsible for generating and checking the WCET certificates. In particular, the checking mechanism is efficiently designed so that code consumers can autonomously verify that the received code meet theirs internal real-time requirements. The certificate generation/checking mechanism is inspired in the Abstraction-Carrying Code framework and implemented using Abstract Interpretation and Linear Programming.

1 Introduction

Embedded systems, in particular real-time systems, often require adaptive configuration mechanisms, where the actualization of available application services, or even operating system services, may be required after their deployment. Traditionally this is done using a manual and heavyweight process specifically dedicated to a particular modification. However, to achieve automatic adaptation of real-time systems, the system design must abandon its traditional monolithic and closed conception and allows itself to reconfigure. An example scenario would be the upgrade of the control software in an automotive embedded system, where the received patch code is dynamically linked in the system, but only after the verification of some safety criteria so that security vulnerabilities or malicious behaviors can be detected before integration.

The main safety criteria in embedded real-time systems is based on the *worst-case execution time* (WCET) of an application. Given a set of concurrent application tasks, the timeliness of the system depends on its capability to assure that execution deadlines are meet at all times. However, the dependency of the WCET on the hardware mechanisms that increase instruction throughput, such as cache memories and pipelining, will increase the cost and complexity of the WCET computation. Considering that embedded systems typically have limited computing resources, the computational burden resulting from the integration of the WCET analyzer into the trusted computing base would be unacceptable. Existent solutions for this problem are, among others, Proof-Carrying Code (PCC)[15], Typed-Assembly Languages (TAL)[14] and Abstraction-Carrying Code (ACC)[2], which common ground is the use of some sort of *certificates* that carry verifiable safety properties about a program and avoid the re-computation of these properties on the consumer side.

The prime benefit of the certificate-based approach is separation of the roles played by the code supplier and code consumer. The computational cost associated to the determination of the safety properties is shifted to the supplier side, where the certificate is generated. On the consumer side, the safety of the program actualization is based on a verification process that checks whether the received certificate, packed along with "untrusted" program, is compliant with the safety policy. To be effective, the certificate checker should be an automatic and stand-alone process and much more efficient than the certificate generator. Besides the certificate checking time, also the size of the certificates will determine if the code actualization process can be performed in reasonable time. Put simply, the main objective of the "certificate+code" setting is to reduce the part of the trusted computing base which attests the compliance of the received code with the safety policy.

However, the use of verifiable WCET estimations as safety properties imposes new challenges to the verification process because of the nature of the techniques used to compute the WCET. In fact, since embedded microprocessors have many specialized features, the WCET cannot be estimated solely on the basis of program flow. Along the lines of [23], state of the art tools for WCET computation evaluate the WCET dependency on the program flow using *Integer Linear Programming* (ILP), while the hardware dependency of the WCET is evaluated using abstract interpretation. Nonetheless, while these tools are tailored to compute tight and precise WCETs, the emphasis of the verification process is more on highly efficient mechanisms for WCET checking. Therefore, we propose an extension of the abstract interpretation-based framework ACC with an efficient mechanism to check the solutions of the linear programming problem.

The implementation of the ACCEPT's static analyzer follows the guidelines proposed by Cousot in [8] for the systematic derivation of abstract transfer functions from the concrete programming language semantics. Fixpoints are defined as the reflexive transitive closure of the set of transition relations ordered in weak topological order [5]. In practice, fixpoints are computed using a chaotic iteration strategy that recursively traverses the dependency graph until the fixpoint algorithm stabilizes. The abstract evaluation of a programs computes, by successive approximations, an abstract context that associates to every program point an abstract value. For WCET analysis, the abstract context used for microarchitectural analysis is adjoined with an abstract context containing the upper bounds for loops. After fixpoint stabilization, the abstract contexts constitute the abstract interpretation part of the ACC certificate.

Novel contributions introduced by this paper are:

– Inclusion of the WCET checking phase inside the ACC framework using the Linear Programming (LP) duality theory. The complexity of the LP problem on the consumer side is reduced from NP-hard to polynomial time, by the fact that LP checking is performed by simple linear algebra computations.

– The *flow conservation* constraints and *capacity* constraints of the LP problem are obtained as abstract interpretations of the assembly transition semantics. The capacity constraints are automatically computed by the parametric static analyzer, which computes the *program flow analysis* as an instrumented *value analysis*.

– Definition of meta-language capable to express the semantics of different programming languages in a unified fixpoint form by means of algebraic relations. The same meta-program is parameterizable by different denotational abstract transfer functions, which are defined for a given abstract domain.

– Definition of transformation algebra based on the meta-language. Two transformations are defined: the first takes advantage of the compositional design of the static analyzer to compute the effect of a sequence of an arbitrary number of instructions between two program points, therefore reducing the size of certificates; the second transforms all possible programs to sequential programs in order to minimize the checking time.

The remainder of this paper is organized as follows. In Section 2 we introduce the meta-semantic formalism used to express the fixpoint semantics of our approach. The program transformation rules are described in Section 3. The verification mechamism of the ILP component is presented in Section 4. Finally, we conclude after the discussion of related work in Section 5.

2 Fixpoint Semantics

For the purpose of static analysis, program semantics are expressed in fixpoint form, where all the possible transitions allowed in the program are taken into consideration. Two important design aspects of ACCEPT's static analyzer are: (1) the implementation of a parametric fixpoint algorithm that is able to compute fixpoints at different levels of abstractions, provided with the corresponding abstract semantics transformers, and (2) the support for the *functional approach* to interprocedural static analysis [21]. To fulfill these requirements, we propose a relational-algebraic treatment of program semantics, combined with denotational definitions of programming language semantics [19]. The pragmatics of this combination is that we specify the structure of programs using polymorphic relational operators and then simulate this specification by providing denotational definitions as arguments. To this end, we employ a modified version of the two-level denotational meta-language defined in [16].

The two levels of the meta-language distinguish between high-level *compile-time* (*ct*) entities and low-level *run-time* (*rt*) entities. At the higher-level, *meta*-programs are compositionally expressed in relational terms by means of binary relational operators. The main advantage of this approach is that new programs can be obtained throughout the composition of smaller programs, in analogy to graph-based languages. Implemented operators are the sequential composition ($*$), the parallel composition ($||$) and the recursive composition ($\oplus$). At the lower level, semantic denotational transformers of type $rt_1 \underline{\rightarrow} rt_2$ provide the desired effects during the interpretation of the meta-program.

$$ct ::= ct_1 * ct_2 \mid ct_1 \mid\mid ct_2 \mid ct_1 \oplus ct_2 \mid rt \tag{1}$$
$$rt ::= \underline{A} \mid [\underline{A}, \underline{A}] \mid rt_1 \underline{\rightarrow} rt_2 \tag{2}$$

In this way, the same meta-program can be used to compute fixpoints parameterized by the abstract domain $\underline{A}$, but leaving the syntactical objects of a particular interpretation hidden from the upper level. Therefore, the upper level of the meta-language is independent from the programming language [20]. To compute the fixpoint of some program P, we first obtain the nondeterministic transition system $\langle \Sigma, \tau \rangle$ of P, where Σ is a nonempty set of states and $\tau \subseteq \Sigma \times \Sigma$ is a binary transition relation between one state and its possible successors. Afterwards, we instantiate a set of semantic transformers of type $\Sigma \rightarrow \Sigma$, defined by $\lambda s \bullet \{s' \mid \exists s' \in \Sigma : s \tau s'\}$, which specify the effect of a particular transition relation τ. Finally, the meta-program of type $\Sigma \rightarrow \Sigma$ is derived as a refinement of the transition system and given as input to the trace-driven static analyzer.

As an illustrating example, consider the source code in Figure 1(a). At the top is the global WCET (336 CPU cycles) annotated at the main procedure by the back-annotation mechanism of ACCEPT [18]. The procedures factorial and foo will be used throughout the paper to exemplify two different execution patterns, the recursive and loop patterns, respectively. The state labeled transition system of the compiled assembly code is given in Figure 1(b) and the meta-program corresponding to the recursive pattern is given in Figure 1(d). For example, the recursive call to factorial is defined by a *trace* which starts with `[bne 16` $\oplus \cdots$, continues with $\cdots *$ `bl -56` $* \cdots$ and ends with $\cdots$ `cmp r3, #1]` until fixpoint stabilization.

The fixpoint algorithm evaluates the meta-program at trace level, but using the program's structural constructs defined at relational level and the program's functional behavior defined at denotational level. The soundness proof of this semantics projection mechanism can be found in [7]. Additionally, the algebraic properties of the upper level of the meta-language provide the means to the transformation of programs, as will be described in Section 3. The interplay between the meta-semantic formalism with the projection mechanism, the program transformation algebra and program verification is illustrated by Figure 2.

Fixpoint semantics is taken from the least fixed point (*lfp*) of the meta-program. For the program state vector $\Sigma = \langle s_i, s_2, \ldots, s_n \rangle$, defined for a particular program P, and the associated functional abstractions $F = \langle f_1, f_2, \ldots, f_n \rangle$ obtained from P's transition system, the state vector Σ is said to be a fixed-point of $F = \langle f_1, f_2, \ldots, f_n \rangle$ if and only if $f_i(s_1, \ldots, s_n) = s_i$. The computational method used to compute this form of fixpoint equations follows from the Kleene first recursion theorem, where every continuous functional $F : L \rightarrow L$, defined

```
/*wcet=336.0 cpu cycles*/
int main(void) {
  int y = factorial(2);
  return y;
}

int factorial(int a) {
  if (a==1)
    return 1;
  else  {
    a = factorial(a-1);
    return a;
  }
}

int foo (int x)  {
  while (x>0) {
    x--;
  }
  return x;
}
```

(a) Source code example

```
             ...
x11{1}: bl 24                  :x5{0}  [d5]
             ...
x12{1}: mov ip, sp             :x11{1} [d11]
             ...
x18{1}: cmp r3, #1             :x17{1} [d17]
x22{1}: bne 16                 :x18{1} [d18]
x19{1}: bne 16                 :x18{1} [d19]
             ...
x30{1}: b 36                   :n21{1} [d22]
x23{1}: ldr r3, [fp, #-16]     :x22{1} [d23]
x24{1}: sub r3, r3, #1         :x23{1} [d24]
             ...
x11{1}: bl -56                 :x25{1} [d26]
             ...
x31{1}: ldr r3, [fp, #-20]     :x30{1} [d31]
             ...
```

(b) Fragment of the assembly program

$0 \cdots 5$ (11 12 (<u>18</u> 22 $\cdots$ 25 11 $\cdots$ 17) 18 19 $\cdots$
21 30 $\cdots$) $\cdots$

(c) Fragment of the weak topological order

```
··· * (bl 24) * ··· (cmp r3, #1) * [bne 16 ⊕ ( ldr r3, [fp, #-16]) * ··· * (bl -56)
··· * (cmp r3, #1)] * (bne 16) * ··· (bne 36) * (ldr r3, [fp, #-20]) * ···
```

(d) The derived meta-program used as input to the static analyzer

Fig. 1. Illustrating Example

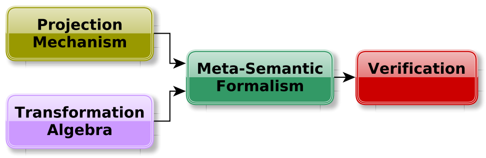


Fig. 2. Different Interactions of the Meta-Semantic Formalism

over the lattice $\langle L, \sqsubseteq, \perp, \top, \sqcup, \sqcap \rangle$, has a least fixed point given by $\bigsqcup_{\delta \geqslant 0} F^{\delta}$, being F^{δ} an ultimately stationary increasing chain ($\delta < \lambda$).

In order to solve data flow equations like $\Sigma = F(\Sigma)$, we apply the *chaotic iteration* strategy [5]. During chaotic iterations, the data flow dependency in the program is taken into consideration, so that the set of program points $i \in C = [1, n]$ are ordered in weak topological order (w.t.o.) [5]. Figure 1(c) shows the w.t.o equivalent to the state labeled transition system in Figure 1(b). For sequential statements, the chaotic algorithm updates exactly one program state per iteration and in the right order. In the case of loops, we apply the *widening* technique pioneered by P. and R. Cousot [9], which consists in choosing a subset of *heads* $W \subseteq C$, (e.g. the set including the underlined label 18 of Figure 1(c)) and replacing each equation $i \in W$ by the equation:

$$s_i = s_i \triangledown f_i(s_1, \ldots, s_n)$$

where "$\triangledown$" is a widening operator.

Then, the *lfp*(F) is computed by the upward abstract iteration sequence:

$$\Sigma^0 = \bot \tag{3}$$

$$\Sigma^{i+1} = \begin{cases} \Sigma^i & \text{if } \Sigma^i = F(\Sigma^i) \\ s_k^{i+1} = s_k^i \triangledown f_k(s_1^i, \ldots, s_n^i) & \forall k = [1, n] : k \in W \\ s_k^{i+1} = f_k(s_1^i, \ldots, s_n^i) & \text{otherwise} \end{cases} \tag{4}$$

The chaotic iteration strategy consists in recursively traversing the dependency graph extracted from the transition system according to a weak topological order. Therefore, chaotic fixpoint iterations mimic the execution order of the program's trace semantics by applying a particular interpretation of the meta-program in some abstract domain. In this way, the execution of a meta-program is in direct correspondence with its fixpoint. As pointed out in [4], widening operators induce strong over-approximations and must be used as less as possible. However, since our WCET analysis is based on the existence of Galois connections between the concrete and abstract domains, we can define the widening ($\triangledown$) as the join ($\sqcup$) in the abstract domain and still be able to compute the least-fixed point when all the program points inside a loop belong the set of widening points W. The reader is referred to [9, Example 4.6] for a detailed explanation.

The design of the fixpoint algorithm in this way has the advantage to allow the computation of path insensitive abstract properties about a program at every program point, but also with the possibility to take into consideration the history of computation as induced by the weak topological order. This is particularly relevant for pipelining analysis, for which the fixpoint algorithm provides an effective method for pipeline simulation (a detailed description of our micro-architectural analysis for the ARM7 [17] target platform can be found at [20]). For this reason, the ACCEPT's static analyzer is able to compute the value analysis of registers and memory location simultaneously with the analysis of cache behavior and the analysis of the pipeline using a generic, parametric and efficient algorithm [19]. When used in the ACC scenario, this feature is of great utility since it provides a one-pass traversal algorithm to check if the certificates behave as fixpoints.

3 Transformation Algebra

The design of the upper level of the meta-language by means of a relational algebra provides a compositional framework to express programs as the composition of elementary semantic building blocks. Each building block is represented by a relation, with the unified functional type $\Sigma \to \Sigma$, regarded as a subgraph. The objective of program transformations is to take advantage of the algebraic properties of the meta-language and reduce the number of connected subgraphs.

In practice, the goal is to reduce the number of program points so that the abstract contexts computed by the static analyzer have a smaller number of entries. However, the transformation must preserve the loop bounds computed for the original program on every program point in order to keep the tightness and soundness properties of the WCET. Thus, the derivation of the meta-program cannot be made directly from the labeled relational semantics, but from an intermediary representation of the control-flow graph, namely a syntax tree structure, which contains the program points necessary to inspect the loops bounds of the original meta-program. The abstract syntax tree of a control-flow graph *CFG* is:

$$CFG \ ::= \ Empty \mid Leaf\ Rel \mid Seq\ CFG\ CFG \mid Par\ CFG\ CFG \mid Rec\ CFG\ CFG$$

In direct correspondence with the upper level of the meta-language, the composition of basic graphical elements are: (1) the sequential composition *Seq* of two subgraphs, (2) the recursive composition *Rec* of two subgraphs, and (3) the parallel composition *Par* of two subgraphs. Apart from the *Empty* graph, the basic element of the *CFG* is a *Leaf* containing a relation *Rel*. According to the projection mechanism presented in Section 2, the *functional abstraction* of a relation is defined by *RelAbs*.

$$\begin{array}{lll} ExecList & ::= & Exec\ Instruction \mid ExecList\ Instruction\ ExecList \\ Rel & ::= & Rel\ (\Sigma, ExecList, \Sigma) \\ RelAbs & ::= & \Sigma \rightarrow \Sigma \end{array}$$

The transformation algebra is based on the compositional properties of relations. Generically, a relation *Rel* can specify an arbitrary number of program transitions, each one defined by an *Instruction*, between any two program points by composing them inside an *ExecList*. The advantage of the functional abstraction is that program effects resulting from the composition of relations are equivalent to those obtained by the functional composition of their abstractions.

With the purpose to reduce the number of program points, we are particularly interested in the simplification of those CFGs composed by two consecutively relations according to the weak topological order. Therefore, candidates for this transformation are instances of subgraphs with type *Seq*, as defined by the function *transfSeq* below. Auxiliary functions are: (1) the function *tail* which provides the last relation *Rel* inside a *CFG*; (2) the function *check* which verifies if the *LoopBounds* in the four points $a, b, c, d \in \Sigma$ are equal so that no loss of program flow information relevant for WCET calculation can occur; and (3) the function *append* which composes two execution lists in one sequence.

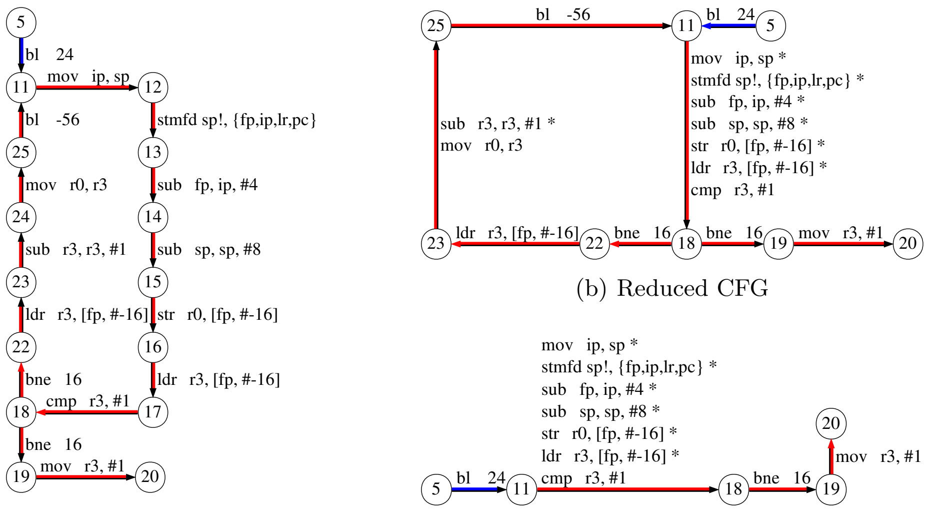
```
transfSeq :: CFG → LoopBounds → CFG
transfSeq (Seq graph (Leaf r)) loops = case tail graph of
    (graph', Just t)     → let Rel (b, it, a) = t
                                Rel (d, ir, c) = r
                            in if check l (a, b, c, d)
                               then let s = Rel (d, append it ir, a)
                                        in transfSeq (Seq graph' (Leaf s)) loops
                               else Seq (transfSeq graph loops) (Leaf r)
```

$(graph', Nothing) \rightarrow\ Seq\ (transfSeq\ graph'\ loops)\ (Leaf\ r)$
$(Empty, Nothing) \rightarrow Leaf\ r$

Now considering the ACC scenario in which the static analyzer also runs on consumer sites, a loop transformation can be applied to the control flow graph so that the fixpoint checking is done within a single state traversal. The ACC program transformation, defined by the function *transfACC*, is based on two facts: (1) the meta-program unrolls the first loop iteration outside the loop subgraph; (2) the static analyzer looks for fixpoint stabilization only at the head of the loop. Therefore, all meta-programs on the consumer side are sequential programs after removing the recursive building blocks:

$transfACC\ \ :: CFG \rightarrow CFG$
$transfACC\ (Rec\ (Leaf\ r)\ graph) = Empty$

Using the intermediate representation *CFG* in combination with the algebraic properties of the meta-language, we now describe the control flow graph transformations in Figure 3 for the example in Figure 1.



(a) Original CFG

(b) Reduced CFG

(c) One-pass CFG

Fig. 3. Examples of Transformed Control-Flow Graphs

The main advantage of transforming the *CFG* in Figure 3(a) into the *CFG* in Figure 3(b), using the function *transfSeq*, is the reduction of program points considered during fixpoint computation and, consequently, so the size of the generated certificate. Additionally, the design of a fixpoint algorithm employing the chaotic fixpoint iteration strategy brings out the possibility to reduce significantly the size of the abstract contexts in some program points. In fact, when applying the fully-sequential transformation of Figure 3(c), using the function *transfACC*, the program points required for fixpoint checking of loops is simply the entry program point of the loop. Therefore, for all the other program points inside the loop, the abstract context consists solely the pipeline state containing the maximal execution count for the corresponding instruction and $\perp$ for the rest of the elements of the abstract domain. For the source example in Figure 1(a), the reduction of the certificate size is shown in Table 1. The structure of the certificates will be described in the next section.

Original Certificate Fig. 3(a)	Sequential Reduction Fig. 3(b)	ACC Reduction Fig. 3(c)	Compressed Certificate (Zip)
13.8 MBytes	5.7 MBytes	5 MBytes	84.2 KBytes

Table 1. Variation of the certificate size

4 Verification

The verification of the WCET is made in different ways on the supplier side and the consumer side. On the supplier side, the verification of the WCET estimates is made at source code level by means of an assertion language based on preconditions and postconditions, which are expressed by our meta-language[20]. Instead of using a deductive system, assertions are evaluated by a relational meta-program which encodes Hoare logic is the following way: "if all the preconditions evaluate to True then if the program output asserts the set of postconditions, then the source code complies with the contract" (termination and soundness is assured by the abstract interpretation framework). The availability of the information about execution times at source level is provided by the back-annotation mechanism of the ACCEPT platform [18]. The `program` being verified and the preconditions (`pre`) and postconditions (`post`) meta-programs are compositionally combined using the relational meta-language in order to obtain the following verification program:

$$\textit{split} * (\texttt{pre} \mid\mid (\texttt{program} * \texttt{post})) * \textit{and})$$

where *split* is an interface adapter that, given the base type $\underline{\text{A}}$, produces $[\underline{\text{A}}, \underline{\text{A}}]$, and *and* is a function that implements logical AND.

The structure ACC certificates generated on the supplier side consists on the abstract contexts (a), computed during the program flow analysis and the micro-architectural analysis, plus the ILP solutions (w) computed by the simplex method on the supplier side. Together with the code (c), the certificate (a, w) is sent to the consumer side as input to the verification mechanism.

On the consumer side, the verification of abstract contexts a is performed by a single one-pass fixpoint iteration over the program c as described in [2], while the ILP checking of w is based on the duality theory [12]. The idea is that to every linear programming problem is associated another linear programming problem called the *dual*. The relationships between the dual problem and the original problem (called the *primal*) will be useful to determine if the received ILP solutions on the consumer side are in fact the optimal ones, that is, the solutions that maximize the WCET objective function on the supplier side.

4.1 The ILP Verification Problem

The optimization problem is defined as the maximization of the objective function *WCET* subject to a set of linear constraints. The variables of the problem are the node iteration variables, x_k, which are defined in terms of the of edge iteration variables, d_{ki}^{IN} and d_{kj}^{OUT}, which correspond to the incoming (i) and outgoing (j) edges to/from a particular program point k contained in the weak topological order $\mathcal{L}$. These linear constraints are called *flow conservation* constraints. Additionally, a set of *capacity constraints* establish the upper bounds (b_{ki} and b_{kj}) for the edge iteration variables.

$$x_k = \sum_{i=1}^{n} d_{ki}^{\mathrm{IN}} = \sum_{j=1}^{m} d_{kj}^{\mathrm{OUT}} \tag{5}$$

$$d_{ki}^{\mathrm{IN}} \leqslant b_{ki} \text{ and } d_{kj}^{\mathrm{OUT}} \leqslant b_{kj} \tag{6}$$

The objective function is a linear function corresponding to the number of node iterations on each program point $k \in \mathcal{L}$, weighted by a set of constants, c_k, which specify the execution cost associated to every program point.

$$WCET = \sum_{k \in \mathcal{L}} c_k . x_k \tag{7}$$

The structure of this optimization problem is particular, in the sense that its solution always assigns integer values to all the variables. This allow us to omit integrality constrains, and furthermore opens the possibility of using linear programming (LP) duality in our approach.

Here, our aim is to demonstrate that the above optimization model can be formally obtained using the theory of abstract interpretation. Note, however, that the *WCET* is not the result of an abstract fixpoint computation. Only the correctness of the LP formulation is covered by abstract interpretation. To this end, the possibility to parameterize the meta-program with different domains is of great importance. The flow conservation constraints are extracted from the program's transition system as an abstraction. For this purpose, the domain of interpretation simply consist on the labels contained in the weak topological order $\mathcal{L}$. Let $\mathcal{T}$ the set of program transitions. Then, the flow conservation constraints (Equation 5) are a set of equations of type $\wp(\wp(\mathcal{T}) \mapsto \mathcal{L})$. Therefore, a Galois connection (α_F, γ_F) can be established between the transition system domain (R) and the flow conservation constraints domain (F) such that:

$$\langle \wp(\mathcal{L} \times \mathcal{T} \times \mathcal{L}), \leqslant \rangle \underset{\alpha_F}{\overset{\gamma_F}{\leftrightarrows}} \langle \wp(\wp(\mathcal{T}) \mapsto \mathcal{L}), \leqslant \rangle$$

$$\begin{aligned}\alpha_F(R) \triangleq &\{x_k = \textstyle\sum_{i=1}^{n} d_{ki}^{\mathrm{IN}} \quad \mid \forall x_k \in \mathcal{L} : d_k^{\mathrm{IN}} = \{e' \mid \exists x_l \in \mathcal{L} : \langle x_l, e', x_k\rangle \in R\}\} \cup \\ &\{x_k = \textstyle\sum_{j=1}^{m} d_{kj}^{\mathrm{OUT}} \mid \forall x_k \in \mathcal{L} : d_k^{\mathrm{OUT}} = \{e' \mid \exists x_l \in \mathcal{L} : \langle x_k, e', x_l\rangle \in R\}\}\end{aligned}$$

$$\begin{aligned}\gamma_F(F) \triangleq \{\langle x_k, d_{\mathrm{out}}, x_l\rangle \mid \quad &\exists s_1 \in F, \exists d_{\mathrm{out}} \in rhs(s_1) : x_k \in lhs(s_1) \;\wedge \\ &\exists s_2 \in F, \exists d_{\mathrm{in}} \in rhs(s_2) : x_l \in lhs(s_2) \quad \wedge \; d_{\mathrm{out}} \equiv d_{\mathrm{in}}\}\end{aligned}$$

The capacity constraints (C) are defined as semantic transformers providing loop bound information. To obtain the loop bounds, we first define the domain of interpretation as an instrumented value domain $\mathcal{V}$ with the loop bounds domain $\mathcal{B}$, and then run the static analyzer. Let program abstract states be $\mathcal{S} = (\mathcal{L} \times \mathcal{V} \times \mathcal{B})$. Then, the semantic transformer f_c, of type $\mathcal{S} \mapsto \mathcal{T} \mapsto \wp(\mathcal{S})$, is obtained as an abstraction of the transition system using the Galois connection (α_C, γ_C):

$$\langle \wp(\mathcal{S} \times \mathcal{T} \times \mathcal{S}), \leqslant\rangle \underset{\alpha_C}{\overset{\gamma_C}{\leftrightarrows}} \langle \mathcal{S} \mapsto \mathcal{T} \mapsto \wp(\mathcal{S}), \sqsubseteq\rangle$$

$$\begin{aligned}\alpha_C(R) &\triangleq \lambda(x_k, v, b) \cdot \{(x_l, v', b') \mid \exists e \in \mathcal{T} : \langle (x_k, v, b), e, (x_l, v', b')\rangle \in R\} \\ \gamma_C(f_c) &\triangleq \{\langle (x_k, v, b), e, (x_l, v', b')\rangle \mid (x_l, v', b') \in f_c(x_k, v, b)\}\end{aligned}$$

The semantic transformer f_c is used by the static analyzer as a run-time entity (see Equation (2) in Section 2). By definition, the transformer f_c increments the loop iterations of a particular transition between the program points $x_k, x_l \in \mathcal{L}$, every time the static analyzer performs a fixpoint iteration over the transition connecting those points. In this way, the static analyzer computes the loop bounds on $\mathcal{B}$, as a side effect of the value analysis on $\mathcal{V}$. For every program point, the last loop iteration computed before the fixpoint stabilization of the value analysis is taken as the upper loop bound.

Verification Mechanism Both the objective function and the set of linear constrains can be represented in matrix form. For this purpose, the node (x) and edge (d) iterations variables are indexed to the variable vector $\mathbf{x}$ of non-negative values. Additionally, the cost values associated to edge variables are zero in the objective function and the edge iterations are zero for all linear equations including a node variable.

The equation system of the *primal* problem is defined in terms of the matrix $\mathbf{A}$, with the coefficients of the constraints (5) and (6), the column vector $\mathbf{x}$ of variables and the column vector $\mathbf{b}$ of capacity constraints. Then, given the row vector $\mathbf{c}$ of cost coefficients, the objective of the primal problem is to maximize the $\mathit{WCET} = \mathbf{cx}$, subject to $\mathbf{Ax} \leqslant \mathbf{b}$. Conversely, the *dual* problem is also defined in terms of the vectors $\mathbf{c}$ and $\mathbf{b}$ plus the matrix $\mathbf{A}$, but the set of dual variables are organized in a complementary column vector $\mathbf{y}$. Then, the objective of the dual problem is to minimize $\mathit{WCET}^{\mathrm{DUAL}} = \mathbf{yb}$, subject to $\mathbf{yA} \geqslant \mathbf{c}$.

Using the simplex method [12], it is possible compute a feasible solution $\mathbf{x}$ for the primal problem and a paired feasible solution $\mathbf{y}$ for the dual problem.

The *strong duality property* of the relationship between this pair of solutions for the purpose of LP checking is: the vector $\mathbf{x}$ is the optimal solution for the primal problem if and only if:

$$WCET = \mathbf{cx} = \mathbf{yb} = WCET^{\,\text{DUAL}}$$

In the ACC setting, this property allows us to use simple linear algebra algorithms to verify the LP solutions that were computed using the simplex method. The verification mechanism is composed by three steps:

1. Use the static analyzer to verify the local execution times included the micro-architectural abstract context. If valid, execution times are organized in the cost row vector $\mathbf{c'}$. Then, take the received primal solutions $\mathbf{x'}$ and solve the equation $WCET' = \mathbf{c'x'}$ to check if it is equal to the received $WCET$.
2. Use the static analyzer to verify the loop bounds abstract context. If valid, loop bounds are organized in the row capacities vector $\mathbf{b'}$. Then, take the received dual solutions $\mathbf{y'}$ and verify the strong duality property by testing the equality of the equation $\mathbf{c'x'} = \mathbf{y'b'}$.
3. Extract the coefficients matrix $\mathbf{A'}$ from the received code and check if the received primal and dual solutions satisfy the equations $\mathbf{A'x'} \leqslant \mathbf{b'}$ and $\mathbf{y'A'} \geqslant \mathbf{c'}$. In conjunction with the two previous steps, this allow us to conclude that $\mathbf{x'}$ and $\mathbf{y'}$ are the optimal solutions of the primal and dual problem and, therefore, conclude that the LP verification is successful.

A numeric example of the LP problem associated to the example in Figure 1 is given in Figure 4. The table in Figure 4(a) shows the primal values and execution costs associates to the LP variables (columns in the matrix $\mathbf{A}$). The variables indexed to x and d are obtained from the labels in Figure 1(b). The linear equation system, from which the coefficients matrix $\mathbf{A}$ are inferred, and the dual values associated to the rows of $\mathbf{A}$ are shown Figure 4(b). Note that the answer to the LP solver will assign to the variable names $x_k \in \mathcal{L}$ the optimal values for node iterations. The vector $\mathbf{b}$ contains the edge iteration upper bounds which are obtained directly from the *program flow* certificate. Provided with this information, the verification mechanism is able to check if the received WCET is in fact the maximal solution of the LP problem, without the need to solve the simplex method all over again.

4.2 Verification Time

The verification time of certificates is strongly reduced for the recursive parts of programs, but not for the purely sequential parts of the program. The reason is that chaotic iteration strategy used during fixpoint computation searches for the least fixed point on the supplier side whereas, in the consumer side, the fixpoint algorithm only verifies if the certificate is one post-fixed point [4].

For a purely sequential set of instructions, chaotic iterations are performed using the third equation in (4), i.e., in the cases where the previous state value

Vars ($\mathbf{x}$)	Primal ($\mathbf{x^*}$)	Costs in CPU cycles ($\mathbf{c}$)
$\cdots$	–	–
x_{15}	4	8
x_{16}	4	7
x_{17}	4	7
x_{18}	4	9
x_{19}	1	10
x_{20}	1	5
x_{21}	1	7
x_{22}	3	10
x_{23}	3	7
x_{24}	3	7
$\cdots$	–	–
d_{18}	3	0
$\cdots$	–	–

(a) Costs and primal values

	Coefficients of variables (matrix $\mathbf{A}$)		Constants ($\mathbf{b}$)	Dual ($\mathbf{y^*}$)
Flow Conservation	$\cdots$	$=$	–	–
	$x_{16} - d_{16}$	$=$	0	0
	$x_{16} - d_{17}$	$=$	0	0
	$x_{17} - d_{17}$	$=$	0	0
	$x_{17} - d_{18}$	$=$	0	0
	$x_{18} - d_{17}$	$=$	0	0
	$x_{18} - d_{18} - d_{19}$	$=$	0	-9
Capacities	$\cdots$	$\leqslant$	–	–
	d_{16}	$\leqslant$	1	19
	d_{17}	$\leqslant$	2	-29
	d_{18}	$\leqslant$	2	29
	d_{19}	$\leqslant$	1	-37
	d_{20}	$\leqslant$	1	37
	d_{21}	$\leqslant$	1	-42
	$\cdots$	$\leqslant$	–	–

(b) Linear equation system and dual values

Fig. 4. Numeric example of the LP problem in matrix form

in the certificate is equal to $\perp$. In such cases, the transition function is computed exactly once for each of the instructions. On the other hand, during the verification of the certificate, the fixpoint stabilization condition will compare the abstract values contained in the received certificate with the output of the single fixpoint iteration running on the consumer side, in order to check if the certificate is a valid post-fixed point. Consequently, the comparison with $\sqsubseteq$ between two states values different from $\perp$ will take longer to compute than the equality test of one state with $\perp$.

For a recursively connected set of instructions, the verification time can be strongly reduced by the fact that the state traversal inside the loop is performed within a single one-pass fixpoint iteration. Two factors contribute for this reduction: (1) the time necessary to compute a valid post-fixed point is much shorter than the time required to perform loop unrolling on the supplier side; (2) with the chaotic iteration strategy, fixpoint iterations over loops are performed only at the *head* of the loop.

Experimental results concerning the checking time of the example in Figure 1 are given in Table 2 (these results were obtained off-device using an Intel®Core2 Duo Processor at 2.8 GHz). The first parcel is relative to the fixpoint algorithm and the second parcel is relative to the LP equation system. The checking time of the solutions of the LP linear system is close to zero in all cases due to the reduction of the LP complexity to polynomial time on the consumer side. As explained before, the performance of the static analyzer is actually worse when the number of instructions outside a loop is significantly bigger compared to the number of instructions inside loops. For the source code example in Figure 1(a), when invoking the function `factorial(4)`, this is specially noticed also due to the sequence of instructions that constitute the epilogue of the recursive function `factorial`. However, when invoking the function `foo` in the `main` procedure, we observe greater reductions of checking time in relation to the generation time for an increasing number of loop iterations.

Function Call	Generation Time (sec)	Verification Time (sec)	Ratio (%)
factorial (4)	1.367 + 0.540	1.942 + 0.004	142.0
foo (3)	1.283 + 0.006	1.013 + 0.005	78.9
foo (7)	3.660 + 0.010	2.160 + 0.003	59.0
foo (15)	14.613 + 0.008	4.495 + 0.012	30.7

Table 2. Experimental Results

5 Related Work

Certifying and checker algorithms using linear programming duality to provide a witness of optimality have been recently proposed by McConnell *et. al.* [13]. In ACCEPT, we complement this approach with a formal definition of the certifying algorithm, using induced abstract interpretations which are correct by construction, and with a formal method to derive the linear programming system of equations by means of Galois connections. The resulting complete algorithm is a *strongly* certifying algorithm for the reason that local execution times are computed using static analysis methods, which always determine sound properties about programs for any possible input values. The *simplicity* and *checkability* of the verification is guaranteed by the fixpoint algorithm, which is exactly the same algorithm on both supplier and consumer sides, plus the strongly duality theory, which enables the checker to run in linear time on the consumer side.

The application of ACC to mobile code safety has been proposed by Albert *et al.* in [2] as an enabling technology for PCC, a first-order logic framework initially proposed by Necula in [15]. One of the arguments posed by Pichardie *et al.* [4] in favor of PCC was that despite its nice mathematical theory of program analysis and solid algorithmic techniques, abstract interpretation methods show a gap between the analysis that is proved correct on paper and the analyzer that actually runs on the machine, advocating that with PCC the implementation of the analysis and the soundness proof are encoded into the same logic, fact that gives rise to a *certified* static analysis. Another relevant research project aiming at the certification of resource consumption in Java-enabled mobile devices is *Mobility, Ubiquity and Security* (MOBIUS) [3]. The *logic-based verification* paradigm of PCC is complemented with a *type-based verification*, whose certificates are derived from typing derivations or fixed-point solutions of abstract interpretations. The general applicability of these two enabling technologies depends on the security property of interest.

In our case, the ACC's certification mechanism based on abstract safety properties is combined with the construction of abstract interpreters which are "correct by construction", as described in [8]. The essential idea is that one abstract interpretation of a program is a formal specification *per se*, which can be induced from the standard interpretation (see [9, Example 6.11] for an use case on the denotational setting). The correctness of abstract interpretations is given by the relation between abstract values in $D^\sharp$ and concrete values in D, often a Galois connection $\langle \wp(D), \alpha, \gamma, D^\sharp \rangle$, where α and γ are the abstraction and concretization functions, respectively. Given the complete join morphism

$F^\natural : \wp(D) \mapsto \wp(D)$, a *correct approximation* $F^\sharp = \alpha \circ F^\natural \circ \gamma$ is obtained by calculus. Then, the fixpoint of $F^\sharp$ is an overapproximation of the fixpoint of $F^\natural$.

Recently in [6], Pichardie *et al.* have presented a certified denotational interpreter which implements static analysis correct by constructions but satisfying the soundness criteria only, leaving aside the problem of precision of the analysis (determined by the abstraction function α). In fact, [9, Section 6] shows that even when the *best* abstraction function is not available, the derivation of abstract interpreters using the algebraic properties of Galois connections is still possible by making $\alpha(d) = \bigsqcup\{\beta(d) \mid d \in D\}$, using the *representation* function β as the singleton set. Notwithstanding, widening operators may be necessary so that the fixpoint in the abstract domain is efficiently computed. In this way, the precision and efficiency of the analysis needs to be balanced in terms of the available α, but that will not compromise the correct construction of abstract interpreters by calculus. Finally, our choice to express concrete programming language semantics in the denotational setting is intentionally associated to the highly declarative programming language Haskell used to implement the ACCEPT static analyzer [20]. Indeed, the induced abstract interpretations are in direct correspondence with the Haskell code implementing them, fact that contributes to the elimination of the gap mentioned in [4].

In ACC [2], verification conditions are generated from the abstract semantics and from a set of assertions in order to attest the compliance of a program with respect to the safety policy. If an automatic verifier is able to validate the verification conditions, then the abstract semantics constitute the certificate. The consumer implements a defensive checking mechanism that not only checks the validity of the certificate w.r.t. the program but also re-generates a trustworthy verification conditions. Conversely, the abstract safety check in PCC is performed by a first order predicate that checks in the abstract domain if a given safety property is satisfied by the reachable states of the program. If the abstract check succeeds then the program is provably safe, otherwise no answer can be given.

In ACCEPT, program safety is expressed in terms of the WCET and since static analysis is not sufficient to compute the WCET, it is not possible to check for WCET safety in the abstract domain. WCET verification in performed in different ways on the supplier and consumer sides. On the supplier side, the ACCEPT platform provides an assertion language to specify the timing behavior at source level [20]. Besides the specification of the permitted WCET, are supported assertions on abstract properties computed at assembly level which are available to the source level by means of a back-annotation mechanism. On the consumer side, WCET verification is performed only at machine code level. Hence, the ACCEPT platform complements the analysis of source code provided by ACC with the analysis at object level provided by PCC.

Another feature of ACCEPT is the possibility to use the meta-language and the parameterizable fixpoint semantics to compute abstract invariants regardless of the programming language, i.e. source or assembly, and then correlate them using compiler debug information (DWARF) [22]. Other approaches combining cost models at source-level and analysis at machine-level to yield verifiable guarantees of resource usage in the context of real-time embedded systems is presented by Hammond *et. al.* in [11], where the static determination of the WCET is performed using the AbsInt's approach [10]. In ACCEPT, the strong requirements concerning resource time and energy consumption of embedded systems compelled us to devise a completely new static analyzer. Comparatively with AbsInt's **aiT** tool, the main feature is the possibility to compute fixpoints at machine-level that *simultaneously* carry out the value analysis, the cache analysis, the pipeline analysis and the program flow analysis.

Finally, Albert *et al.* present in [1] a fixpoint technique to reduce the size of certificates. The idea is to take into account the data flow dependencies in the program and actualize the fixpoint only at the program points that have their predecessors states *updated* during the last iteration. In ACCEPT, the same notion of certificate-size reduction is achieved by means of a program transformation algebra combined with loop unrolling and fixpoint chaotic iterations. The chaotic iteration strategy allows the fixpoint algorithm to look for stabilization at the entry-point of loops for the whole loop to be stable [5]. By the fact that when using meta-language, the first loop iteration is unrolled outside the loop, we apply a program transformation to loop structures that consists in transforming programs with loops in purely sequential programs by keeping only the entry-points of loops.

6 Conclusions

This paper reports the application of a compositional static analyzer based on abstract interpretation used to compute the WCET of a program in the context of Abstract-Carrying Code (ACC). The novelty of the approach consists in using the WCET as the safety parameter associated to a verification mechanism that is able to check that the ACC certificates are valid within a one-pass fixpoint iteration and then check if the WCET is correct using the duality theory applied to linear programming. Experimental results show that, for highly sequential programs, the computation of the least fixed point can be more efficient than one single iteration over a post-fixed point. Therefore, the verification process is only efficient when in presence of highly iterative programs.

Besides the reduction of verification times, the concept of ACC also requires methods to reduce the size of certificates. We have presented a transformation algebra that applies to control flow graphs in order to minimize the number of program points considered during fixpoint computations. The simplicity of the process relies on the algebraic properties of the meta-language and on the compositional design of the chaotic fixpoint algorithm.

As prospective future work, we intend to extend the timing analysis and certification of concurrent applications running on multi-core environments. The main challenge is to extend the usage of the meta-language to model architectural flows as well as the application's control flow. The type system of the upper level of the meta-language will allow the integration of these two levels of abstraction in a unified program representation. Therefore, the generic applicability of the constructive fixpoint algorithm and, consequently, of the certification mechanism, is guaranteed.

7 Acknowledgments

This work is partially funded by LIACC, through the Programa de Financiamento Plurianual, FCT, and by the FAVAS project, PTDC/EIA-CCO/105034/2008, FCT.

References

1. Elvira Albert, Puri Arenas, Germán Puebla, and Manuel V. Hermenegildo. Certificate size reduction in abstraction-carrying code. *CoRR*, abs/1010.4533, 2010.
2. Elvira Albert, Germán Puebla, and Manuel Hermenegildo. An abstract interpretation-based approach to mobile code safety. *Electron. Notes Theor. Comput. Sci.*, 132(1):113–129, 2005.
3. Gilles Barthe, Lennart Beringer, Pierre Crégut, Benjamin Grégoire, Martin Hofmann, Peter Müller, Erik Poll, Germán Puebla, Ian Stark, and Eric Vétillard. Mobius: Mobility, ubiquity, security. In *TGC*, pages 10–29, 2006.
4. Frédéric Besson, David Cachera, Thomas Jensen, and David Pichardie. Certified static analysis by abstract interpretation. In *Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures*, pages 223–257, Berlin, Heidelberg, 2009. Springer-Verlag.
5. François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *In Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer-Verlag, 1993.
6. David Cachera and David Pichardie. A certified denotational abstract interpreter. In *Proc. of International Conference on Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 9–24. Springer-Verlag, 2010.
7. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Electronic Notes in Theoretical Computer Science*, 6, 1997.
8. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design.* NATO ASI Series F. IOS Press, Amsterdam, 1999.
9. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2:511–547, 1992.
10. Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise wcet determination for a real-life processor. In *Proceedings of the First International Workshop on Embedded Software*, EMSOFT '01, pages 469–485, London, UK, 2001. Springer-Verlag.
11. Kevin Hammond, Christian Ferdinand, Reinhold Heckmann, Roy Dyckhoff, Martin Hofmann, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert F. Pointon, Norman Scaife, Jocelyn Sérot, and Andy Wallace. Towards formally verifiable wcet analysis for a functional programming language. In *WCET*, 2006.
12. Frederick S. Hillier and Gerald J. Lieberman. *Introduction to operations research, 4th ed.* Holden-Day, Inc., San Francisco, CA, USA, 1986.

13. Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.
14. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21:527–568, May 1999.
15. George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.
16. Hanne Riis Nielson and Flemming Nielson. Pragmatic aspects of two-level denotational meta-languages. In *Proceedings of the European Symposium on Programming*, ESOP '86, pages 133–143, London, UK, 1986. Springer-Verlag.
17. Vishnu Patankar, Alok Jain, and Randal Bryant. Formal verification of an arm processor. In *12th International Conference On VLSI Design*, pages 282–287, 1999.
18. Vítor Rodrigues, Mário Florido, and Simão Melo de Sousa. Back annotation in action: from wcet analysis to source code verification. In *Actas of CoRTA 2011: Compilers, Prog. Languages, Related Technologies and Applications*, July 2011.
19. Vítor Rodrigues, Mário Florido, and Simão Melo de Sousa. A functional approach to worst-case execution time analysis. In *20th International Workshop on Functional and (Constraint) Logic Programming (WFLP)*, pages 86–103. Springer, 2011.
20. Vítor Rodrigues, Mário Florido, and Simão Melo de Sousa. Towards adaptive real-time systems by worst-case execution time checking. Technical report, Artificial Intelligence and Computer Science Laboratory (LIACC)- University of Porto, 2011.
21. Micha Sharir and Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*, pages 189–233. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.
22. The DWARF Debugging Standard . `http://www.dwarfstd.org/`.
23. Reinhard Wilhelm. Why ai + ilp is good for wcet, but mc is not, nor ilp alone. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 309–322. Springer Berlin / Heidelberg, 2003.