2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications

Improving WCET Analysis Precision through Automata Product

Vincent Mussot* and Pascal Sotin*
*Toulouse University, IRIT[1]
Email: {mussot, sotin}@irit.fr

***Abstract*—Real-time scheduling of application requires sound estimation of the Worst-Case Execution Time (WCET) of each task. Part of the over-approximation introduced by the WCET analysis of a task comes from not taking into account the fact that the (implicit) worst-case execution path may be infeasible.**

This article does not address the question of finding infeasible paths but provides a new formalism of automata to describe sets of infeasible paths. This formalism combines the possibilities to express state-based path acceptance (like in regular automata), constraints on counters (in the Implicit Path Enumeration Technique fashion) and contexts of validity (like in Statecharts).

We show the applicability of our proposal by performing infeasible paths aware WCET analyses within the OTAWA framework. We provide algorithms that transform the control flow graph and/or the constraints system supporting the WCET analysis in order to exclude the specified paths.

I. Introduction

a) Context: Task scheduling and resource management lie at the very heart of embedded systems. Under-sizing hardware may lead to unsafe and erroneous executions of programs while over-sizing will increase costs. As a result, a major aspect of the real-time domain is to get as close as possible to the optimal dimensions for every single element of the architecture while remaining safe.

A critical information needed to dimension the whole system is a Worst-Case Execution Time (WCET) for each task. A WCET is an upper bound on the execution time of a task. Although it is hardly possible to determine the exact WCET of a program, static timing analysers can compute a safe upper bound instead.

Timing analysis relies on both low-level and high-level analyses which work together in an effort to precisely estimate the timing of a task. Low-level analysis focuses on the behaviour of caches, on the processor pipeline and on the memory accesses to determine the execution time of groups of instructions. High-level analysis combines the basic timings according to the program control flow. A popular way of doing that is the Implicit Path Enumeration Technique (IPET). The low-level timings and the dependencies between number of executions of each basic block are turned into an Integer Linear Program (ILP). Its resolution delivers a safe WCET.

b) Motivations: The path implicitly considered by IPET may be infeasible, meaning that it does not correspond to any realistic execution of the program. Infeasible paths are a major challenge for WCET analysis since they can cause serious WCET over-estimation (see experiments in Section V)[1].

[1]This work is supported by the French ANR project W-SEPT.

If we take an arbitrary path in the Control Flow Graph (CFG) of a program, it might be infeasible for reasons like:

i. It executes some dead code (undetected by the compiler),
ii. It iterates a loop too many times,
iii. It goes through two exclusive branches,
iv. It executes some code in a context which makes it dead code…

IPET-based static analysers, according to what they can detect or what they are told, try to reflect the infeasibility by either modifying the CFG (Item i. typically) or enriching the ILP (Item ii. typically). We do not discuss in this article the question of infeasibility detection but focus on expressing infeasible paths and integrating automatically and faithfully this information in the WCET analysis.

Annotation languages for WCET analysis deal with the question of expressing facts (for a complete survey, see [1]) but all suffer at least one of the following limitations:

- The expressiveness is limited to ease the integration,
- The computational cost of the integration cannot be controlled,
- The integration is reduced to adding new ILP constraints,
- Part of the annotation that can be written cannot be integrated in the analysis process,
- The integration process has not been formalised.

c) Contributions: Our contribution is not a new WCET annotation language but a versatile automata-based formalism to describe infeasible paths. Our automata are first class citizens:

- An automaton recognizes a well-defined language of CFG transitions (Section III-B).
- They can be built using a combination of well-known features in the field of static analysis: linear constraints, state-based acceptances and contexts of validity (Section III-A).
- We formally define operations for automata manipulation (Section III-C) and provide the corresponding algorithms (Section IV), including a product with any CFG.
- We show the applicability of our proposal by implementing it as a plug-in for OTAWA, an academic state-of-the-art framework for WCET analysis (Section V).

We illustrate our purpose in Section II with an example that shows how path properties are expressed in our formalism and how this information is integrated in the analysis process. In Section VI we discuss the strengths and limitations of our proposal, relating it to existing works.

II. Overview

In this section, we give an example where infeasible paths are damaging the WCET analysis, and solve this issue using the formalism and algorithms we propose. Here is the process:

- Somebody or something identifies that a program contains syntactic paths that are not feasible (Section II-A),
- An automaton that encodes the source of the infeasibility is created. We call it a Path Property Automaton (PPA) (Section II-B),
- The program CFG and the PPA are merged into a product PPA that accepts only feasible syntactic paths (Section II-C),
- The resulting PPA is brought back to a representation suitable for a WCET analysis tool, possibly at the cost of an over-approximation of the feasible paths (Section II-D).

A. Software

We consider a fragment of software containing a loop whose maximal number of iterations (N) is known. The loop body contains two `if-then` statements. The compilation process turned this piece of code into a binary file whose CFG, built by a static analysis tool, is depicted on Figure 1. Nodes of this graph represent basic blocks of assembly instructions. Edges of this graph represent transitions between blocks due to code continuation or branching instructions. The static analysis tool estimated a WCET which happens to be seriously above the measured timings.

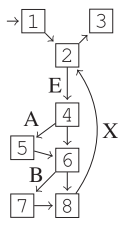

Fig. 1. CFG.

The analysis of the source code, either by a human or by a static analyser, revealed that in each iteration of the loop, at most one `then` branch is executed. Since that fact does not appear on the CFG, IPET maximized the WCET by picking A and B in each iteration of the loop. In a better world, the programmer would have used an `if-then-else` statement or a static analysis within the compiler or the WCET tool would have noticed and exploited that fact. Unfortunately, in the real world, WCET static analysers have no control on the source code nor on the compilation process, and their first concern is to model faithfully the low level behaviour. Infeasible paths are for them an orthogonal issue. We thus face the problem of telling the WCET analyser, in the least intrusive way, to ignore paths where both transitions A and B are taken during one iteration of the loop.

B. Infeasible paths

Using the formalism proposed in this article (detailed in Section III) we encode the exclusivity property mentioned before as a Path Property Automaton (PPA) that rejects the infeasible paths. On Figure 2, we show two alternative encodings. On both PPAs, the CFG transitions E (entry) and X (exit) define the boundaries of an iteration. Within this context (rounded corners rectangle) the number of CFG transitions A and B are counted. On Figure 2a the counting process relies on a counter attached to the transitions and a constraint attached to the context: when leaving the context, the α counter must be

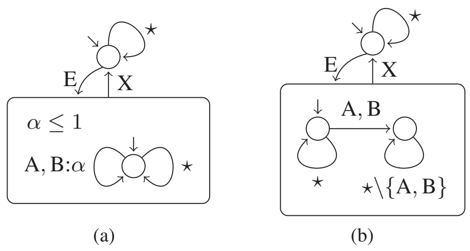

Fig. 2. Two PPAs rejecting the infeasible paths.

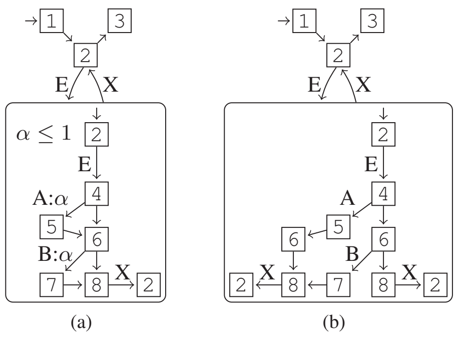

Fig. 3. Two PPAs accepting the feasible syntactic paths.

lower or equal to 1. The $\star$ symbol denotes a default transition. On Figure 2b counting is based on states: once either A or B has been seen, both are no longer accepted.

C. Product

The CFG of Figure 1 can be considered as a PPA that describes the syntactic paths of the software. Hence we now reduce our concerns to combine the CFG and the property in a PPA accepting only paths that are both syntactically valid and feasible (w.r.t. the property). In automata theory, This operation is called a product. We do not define a generic product for any pair of PPAs but exploit the flat nature of the CFG to define an asymmetric product that we call *injection product*. We call the PPA resulting from this operation a hierarchical CFG. Figures 3a and 3b respectively result from the injection of the CFG in 2a and 2b. Note that some basic blocks may be duplicated in the internal representation of the WCET analysis tool but that the program itself is unchanged.

D. Flattening

Both automata shown on Figure 3 describe exactly the feasible paths of our software. Unfortunately, WCET analysers are not used to this kind of hierarchical objects. We thus need an operation to get back to a flat PPA. We call this operation a *flattening*. Figures 4a and 4b respectively results from the flattening of 3a and 3b. On Figure 4a the CFG carries a numerical constraint, entailed by the original one, that prevents from always taking A and B in all iterations. On Figure 4b the possibility of executing both A and B in the same iteration has been syntactically removed.

We give back the CFG of Figures 4a and 4b to the WCET analyser which respectively computes a 26% and a 40% more

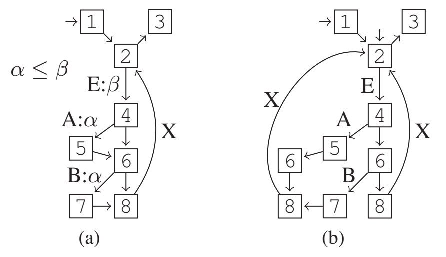

Fig. 4. Two result CFG.

precise WCET. The gap between the two evaluations is due to the fact that the CFG transformation allowed some low level analysis improvement. Details and additional experiments are presented in Section V.

III. Formalism

In this section we present an automata formalism to encode feasible paths (and by complement infeasible paths). This proposal is novel and is directed toward an efficient integration of the feasibility information in the IPET-based WCET analyses.

A. Features

We present four features offered by the formalism allowing concise and modular expression of a set of paths.

1) Nodes and arrows: An automaton can be built using nodes and arrows[2]. One node is marked as initial. To each arrow is attached a set of program transitions[3]. A program transition is the transfer of the instruction pointer from a control point to another. The $\star$ symbol is used to accept any program transition which is not accepted by any other outgoing arrow.

A path is accepted by the automaton if its program transitions can be consumed by the automaton starting in the initial state. We consider deterministic automata where a transition is carried by at most one outgoing arrow of each node.

Figure 5a shows an automaton that accepts any trace provided that this trace does not contain the program transition `0x10` $\mapsto$ `0x12` (i.e. this program transition is infeasible). The automaton of Figure 5b accepts at first any transition but refuses transition F once the program went through transition E (i.e. transition E makes transition F infeasible).

2) Counters and constraints: In addition to sets of program transitions, the arrows of an automaton can carry a set of counters. Throughout the article, we use Greek letters to denote a counter. These counters are used in a conjunction of linear constraints attached to the automaton itself.

An execution path is accepted if it is accepted by the nodes and arrows of the automaton and if the number of occurrences of each counter on the path satisfies the constraint.

[2]The consecrated words in automata theory are *states* and *transitions*. We keep these words for more semantic notions (CFG transition, current state in an automaton) and use instead *nodes* and *arrows* to denote the syntactic elements that compose an automaton.

[3]More generally each arrow carries a set of symbols taken in an alphabet. In our setting we chose the alphabet to be all the program transitions.

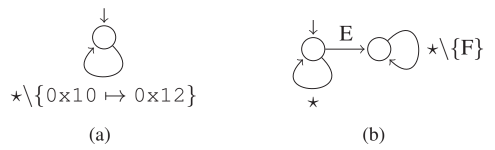

Fig. 5. Nodes and Arrows: examples

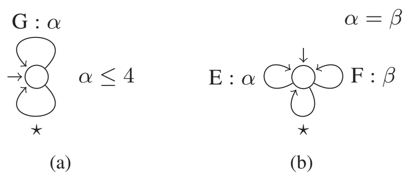

Fig. 6. Counters and Constraints: examples

Figure 6a denotes the set of paths where the program transition G is taken at most four times. Figure 6b denotes the set of paths where transition E is taken the same number of times as transition F. The latter example illustrates a major difference between our proposal and the counter automata [2]. Using regular counter automata, a program transition is accepted only if the *current* state of the counters allows it. Our constraint only applies when considering the whole path: eventually $\alpha = \beta$.

3) Super-node and sub-automaton: Each node of a PPA may contain another automaton. The node is then called a super-node and the embedded automaton is called its sub-automaton. This turns our proposal into a formalism similar to the hierarchical automata [3], [4].

The acceptance rules are modified as follows:

- When the (parent) automaton consumes a program transition and ends up on a super-node, its sub-automaton must also consume the transition (starting from its initial state). The new state is composed of the current super-node together with the current state of the sub-automaton.
- When the (parent) automaton is currently on a super-node and the transition to consume is not carried by any outgoing arrow then the sub-automaton must consume the transition. The parent automaton stays on the super-node and the state of the sub-automaton is updated.
- When the (parent) automaton is currently on a super-node and the transition to consume is carried by an outgoing arrow then the sub-automaton must consume the transition (still). The parent automaton then consumes the transition and ends up on the head of the arrow.

In other words reaching a super-node temporarily gives the control to the sub-automaton. The path fragment starting by the program transition leading to the super-node and ending with the first transition that leaves the super-node must be accepted by this sub-automaton. One should note the overlapping on the first and last transitions, and the priority given to the parent arrows. A program transition can thus be used to enter and/or exit multiple levels of hierarchy at once.

When leaving a super-node, the path fragment accepted by its sub-automaton must satisfy the numerical constraints on the counters of this automaton. This counters are forgotten afterwards (they are “reset” in some sense).

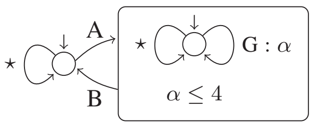

Fig. 7. Super-node and Sub-automaton: example

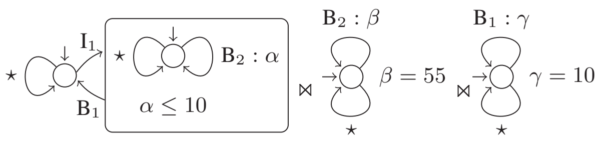

Fig. 8. Example of a Symbolic Product

Figure 7 states that after a transition A of a path is encountered, at most four transitions G are traversed before the next transition B.

4) Symbolic product: In our formalism, an automaton can be a hierarchical automaton carrying constraints, as presented in the beginning of this section but it can also be the symbolic product of several automata. A path accepted by all the automata of a symbolic product is accepted by the symbolic product itself. The symbolic product can be read as a conjunction of properties or as an intersection of languages. If $\mathcal{A}$ and $\mathcal{B}$ are two PPAs, then $\mathcal{A} \bowtie \mathcal{B}$ denotes their symbolic product.

Figure 8 provides information on an inner loop whose back edge is B$_2$, nested in an outer loop whose body starts by the program transition I$_1$ and ends with its back edge B$_1$. The inner loop presents a triangular behaviour. Its maximal number of iterations in an iteration of the outer loop is 10 (as stated by the first hierarchical automaton) but it happens that on a complete execution of the outer loop the total number of iterations of the inner loop is exactly 55 (second automaton). The third automaton provides an exact bound for the outer loop. The symbolic product of Figure 8 could eventually be embedded in the context of the outer loop.

B. Recognized language

We give here a formal definition of the set of words accepted by an automaton built using the features presented in III-A. We write $[\![\mathcal{A}]\!] \in \mathcal{P}(\Sigma^\star)$ for the set of words accepted by the automaton $\mathcal{A} \in \mathit{Auto}$. The set Σ is an alphabet; in our instantiation, it contains all CFG transitions.

An automaton can be either a symbolic product or a hierarchical automata ie. $\mathit{Auto} = \mathit{Prod} \uplus \mathit{Hier}$. The language of a symbolic product is defined by:

$$[\![\mathcal{A}_1 \bowtie \cdots \bowtie \mathcal{A}_n]\!] = [\![\mathcal{A}_1]\!] \cap \ldots \cap [\![\mathcal{A}_n]\!]$$

If $\mathcal{A}$ is not a symbolic product then it is a hierarchical automaton defined by the tuple $\langle N, i, \mathit{sub}, X, \Phi, \mathit{Arr}\rangle$ where:

- N is a set of nodes and $i \in N$ is the initial one,
- $\mathit{sub} \in N \to (\mathit{Auto} \uplus \{\perp\})$ associates a sub-automaton to each super-node and $\perp$ to the ordinary nodes,
- X is a set of counters (Greek letters in the constraints),
- Φ is a conjunction of linear constraints over X,
- *Arr* is a set of arrows. An arrow is defined by its source node, its destination node, a set of accepted transitions, and a set of counters to increment when crossing it.

The presence of a hierarchy makes the acceptance definition recursive. To handle the top level of the hierarchy, we define the set $\mathit{Walk}(\mathcal{A})$ of valid walks in $\mathcal{A}$. A valid walk π starts at node i and follows arrows in *Arr* (walks do not descend in the hierarchy). We define the following notations:

- $\pi \models \Phi$ states that the counters seen on π satisfy Φ,
- $|\pi|$ is the number of arrows traversed by π,
- $\mathit{acc}_k(\pi)$ is the set of transitions accepted by the k-th arrow of the walk π,
- $\mathit{node}_k(\pi)$ is the k-th node of the walk π.

The set of words accepted by the hierarchical automaton $\mathcal{A}$ is the set of words $t \in \Sigma^\star$ such that:

$$\begin{array}{l}
\exists \pi \in \mathit{Walk}(\mathcal{A}), |\pi| = m, \pi \models \Phi,\\
\exists a \in \Sigma^m, s \in (\Sigma^\star)^{m+1}, t = s_0.a_0 \ldots s_{m-1}.a_{m-1}.s_m,\\
\forall k, a_k \in \mathit{acc}_k(\pi)\\
\forall k, \mathit{node}_k(\pi) = n,\\
\quad \left\{ \begin{array}{lll} \mathit{sub}(n) = \mathcal{A}' & \Rightarrow & \left\{ \begin{array}{l} a_k.s_k.a_{k+1} \in [\![\mathcal{A}']\!] \\ \text{outgoing}(n) \cap (\downarrow s_k) = \emptyset \end{array} \right. \\ \mathit{sub}(n) = \perp & \Rightarrow & s_k = \varepsilon \end{array} \right.
\end{array}$$

where x_i is the element at position i (starting from 0) in the sequence x; a_{-1} and a_m are defined to be the empty word ε; the dot operator stands for the concatenation; $\downarrow u$ stands for the set of letters of the word u; outgoing(n) stands for the letters accepted by the arrows leaving node n.

In the formula above, t is decomposed on two levels according to the walk π. The sequence of *letters* a is made of single transitions consumed by the arrows of the automaton itself. The sequence of *words* s is made of fragments of words matched by the sub-automata contained in the nodes. The formula enforces the priority of the automaton on the sub-automata and the consumption of transitions at several levels (as described in Section III-A3).

C. Operations on automata

Both the program CFG and infeasible paths can be encoded in our formalism. A CFG is represented by an isomorphic automaton $\mathcal{C}$ where every basic block becomes a node and every edge becomes an arrow labelled by the edge (eg. Fig 1). The infeasible paths are encoded as an automaton $\mathcal{P}$ that accepts any paths but the infeasible ones (eg. Figs 2,5,6,7,8).

Using operations on automata, we combine $\mathcal{C}$ and $\mathcal{P}$ into an automaton $\mathcal{C}'$ that can be fed to most of the existing IPET-based WCET analysers. We define the set *Flat* of PPAs that can be built using only nodes, arrows carrying a single transition, counters and constraints (Sections III-A1 and III-A2). The *Flat* subset of *Auto* is assumed to be similar to the internal representation of an IPET-based WCET analysers. The operations that we will consider are:

Name	Symbol	Type
Product	$\times$	$\mathit{Flat} \times \mathit{Flat} \to \mathit{Flat}$
Unfolding	*unfold*	$\mathit{Auto} \to \mathit{Flat}$
Flattening	*flatten*	$\mathit{Auto} \to \mathit{Flat}$
Injection	$\rtimes$	$\mathit{Flat} \times \mathit{Auto} \to \mathit{Auto}$

The product is the usual product of automata. The unfolding operations turns an automaton into an equivalent flat one having no counter nor constraints. The flattening operation turns an automaton into a more permissive flat one. The injection is a special kind of product that adapts the transitions of a flat automaton to the structure of another automaton.

Ideally we would like a flat automaton $\mathcal{C}'$ of reasonable size such that $[\![\mathcal{C}']\!] = [\![\mathcal{C}]\!] \cap [\![\mathcal{P}]\!]$. Unfortunately, no combination of the operations mentioned above is fully satisfying and we need to renounce either to:

1) *Flatness*. $\mathcal{C} \rtimes \mathcal{P}$ is a non-flat PPA whose accepted paths are exactly the desired ones. It is not compatible with the existing WCET analysers.
2) *Reasonable size*. The unfolding operation would allow to compute a flat automaton encoding precisely the paths common to $\mathcal{C}$ and $\mathcal{P}$. This method would face serious scalability issues.
3) *Exactness*. In the context of WCET analysis, we can relax our ambition into finding a flat automaton $\mathcal{C}'$ such that $[\![\mathcal{C}']\!] \supseteq [\![\mathcal{C}]\!] \cap [\![\mathcal{P}]\!]$ (safe approximation).

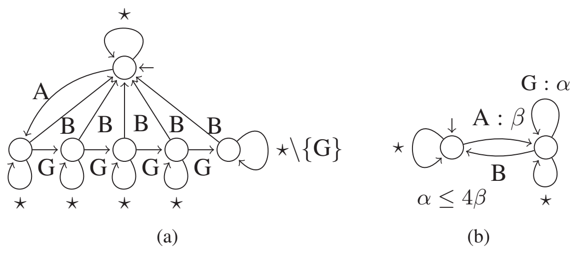

Fig. 9. Unfolding and flattening of Figure 7

Figure 9 illustrates the transformations 2 and 3 on the automaton of Figure 7. If we leave this automaton unchanged, it is not flat (Item 1 lost). If we unfold it as shown on Figure 9a then its size has seriously grown (Item 2 lost) and it would be even worse with a constant of 100 instead of 4. If we flatten it as shown on Figure 9b then it becomes a two-state automaton but the path AGGGGGBAB containing five consecutive G is accepted for the reason that it contains two A (Item 3 lost).

We follow the solution of Item 3 and choose to compute *flatten*$(\mathcal{C} \rtimes \mathcal{P})$. Note that $\mathcal{C} \times$ *flatten*$(\mathcal{P})$ is not always a reasonable option for computing $\mathcal{C}'$. The presence of symbolic products in $\mathcal{P}$ can lead to an automaton *flatten*$(\mathcal{P})$ of exponential size. Performing the injection first then flattening reduces this issue to the only points where it is required to faithfully express the property.

IV. Algorithms

The two main stages of the CFG transformation and enrichment are detailed in this section. The first stage is injecting the CFG into a PPA in order to obtain a hierarchical CFG. This is done by the *InjectionProduct* algorithm (Function 1) which relies on *AcceptTrans* (Function 2). The second stage is flattening the newly built CFG to retrieve a usual flat CFG enclosing the path properties. This is done by the *Flatten* algorithm (Procedure 3) which relies on *BreakFrontier* (Procedure 4) and the classical automaton product. These operations were sketched in Sections II-C and II-D. Their specification was defined in Section III-C.

Function 1: InjectionProduct

```
Data: Flat $\mathcal{C}$, Auto $\mathcal{P}$
Result: Auto $\mathcal{R}$
1  // WorkList WL contains ⟨Node, State, State⟩
2  initialization of the new automaton $\mathcal{R}$
3  create $\mathcal{R}$ initial state from those of $\mathcal{C}$ and $\mathcal{P}$
4  begin with $\mathcal{C}$, $\mathcal{P}$ and $\mathcal{R}$ initial states in the WL
5  foreach Triplet ⟨ $N_\mathcal{C}$, $S_\mathcal{P}$, $S_\mathcal{R}$⟩ of WL do
6      retrieve the list $L$ of outgoing arrows from node $N_\mathcal{C}$
7      foreach Arrow Arr of L do
8          save the current $\mathcal{R}$
9          try
10             pair ⟨$S'_\mathcal{P}$, $S'_\mathcal{R}$ ⟩ = AcceptTrans($S_\mathcal{P}$, Arr, $S_\mathcal{R}$)
11             retrieve the head node $N'_\mathcal{C}$ of Arr
12             push ⟨$N'_\mathcal{C}$, $S'_\mathcal{P}$, $S'_\mathcal{R}$⟩ in the WL
13         catch failure
14             rollback $\mathcal{R}$
15 return $\mathcal{R}$
```

A. Injection product

Function 1 creates a new automaton from the injection of a flat automaton into a PPA that can be hierarchical or a product of automata. This function has been designed to inject a classical CFG into an automaton representing semantic properties with context-sensitivity, in order to create a hierarchical CFG.

This product relies on a work list algorithm which uses the *AcceptTrans* function to build nodes, arrows, sub-automata, and constraints in the result automaton. Upon a successful call to *AcceptTrans*, the result automaton will integrate the changes, while upon a failure it will remain unchanged

In the classical automaton literature, work lists contains states of the automaton (that we call nodes). Due to the hierarchy and the symbolic product in our formalism, the notion of node and state no longer coincide. In our setting, we define a state to be:

- The name of a regular node N in a hierarchical automaton
- The name of a super-node in a hierarchical automaton together with a state in its sub-automaton ($N.S$).
- A tuple of state $\langle S_1 \ldots S_n \rangle$ when the automaton is a symbolic product

Function 2 allows to follow a specific arrow *Arr* in an automaton $\mathcal{A}$ which can be hierarchical or a symbolic product. In this process, it creates the associated states, arrows, sub-automata and constraints in the result automaton. This function takes as parameters the arrow that should be followed, the state in $\mathcal{A}$ where to begin the arrow matching, and the state in the result automaton where to continue the construction. The treatment of the hierarchy follows the acceptance criteria of Section III-A3:

- We descend in the node hierarchy until we find a matching arrow
- The corresponding states and arrows are then built in the result automaton.
- The arrow shall then be accepted by the matching arrow tail sub-automata if it exists.

Function 2: AcceptTrans

```
Data: State $S$, Arrow Arr, State $R$
Result: State $S'$, State $R'$
1  Auto $\mathcal{A}$ = retrieve automaton where $S$ belongs
2  if $\mathcal{A}$ is a Prod then //S,R are of the form $\langle X_1, \ldots, X_n \rangle$
3      foreach automaton $\mathcal{A}_1 \ldots \mathcal{A}_n$ that composes $\mathcal{A}$ do
4          $\langle S'_i, R'_i \rangle$ = AcceptTrans($S_i$, Arr, $R_i$)
5      $S' = \langle S'_1, \ldots, S'_n \rangle$ ; $R' = \langle R'_1, \ldots, R'_n \rangle$
6  else //$\mathcal{A}$ is a Hier; S,R are of the form $N.S_{sub}$, $M.R_{sub}$
7      if Arr matches an outgoing arrow of N then
8          //We note $N'$ the head of $N$ outgoing arrow
9          create or retrieve the node $M'$ in $M$ automaton
           create $Arr_{res}$ by copying Arr in $M$ automaton
10         add counters from both arrows on $Arr_{res}$
11         if N has a sub-automaton sub-$\mathcal{A}$ then
12             AcceptTrans($S_{sub}$, Arr, $R_{sub}$)
13             //This might spread a failure
14         if $N'$ has a sub-automaton sub-$\mathcal{A}'$ then
15             //We note $S_{ini}$ the initial state of sub-$\mathcal{A}'$
16             create sub-automaton sub-$\mathcal{A}_{R'}$ in $R'$
17             create an initial state $R_{ini}$ in sub-$\mathcal{A}_{R'}$
18             $\langle S'_{ini}, R'_{ini} \rangle$ = AcceptTrans($S_{ini}$, Arr, $R_{ini}$)
19             $S' = N'.S'_{ini}$ ; $R' = M'.R'_{ini}$
20         else
21             $S' = N'$ ; $R' = M'$
22     else
23         if N has a sub-automaton sub-$\mathcal{A}$ then
24             $\langle S'_{sub}, R'_{sub} \rangle$ = AcceptTrans($S_{sub}$, Arr, $R_{sub}$)
25             $S' = N.S'_{sub}$ ; $R' = M.R'_{sub}$
26         else
27             raise failure
28 return $\langle S', R' \rangle$
```

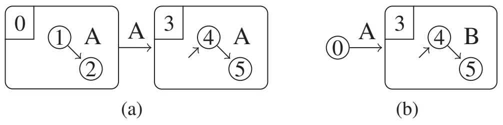

Fig. 10. Illustration of AcceptTrans

- Finally the arrow shall be accepted by the matching arrow head sub-automata if it exists and a new automaton in the result automaton should then be created.

Figure 10a represents a case where *AcceptTrans* was called with the state 0.1 as start state, and the arrow *A*. In this case, the parent automaton will try to consume the arrow from its super-node 0 to its super-node 3. Prior to this consumption, since 0 was a super-node, its sub-automata must accept the arrow, and that is done by taking the arrow from 1 to 2 in this sub-automaton. Once the parent automaton has consumed the arrow, it reaches the super-node 3 which must then accept the arrow starting from its initial sate 4, which is done by going from 4 to 5 in this sub-automaton. In the end, the arrow *A* accepted by this automaton from the state 0.1 ends up in the state 3.5.

Figure 10b represents the failure case where an arrow *A* is not accepted by the sub-automaton of its ending node.

Procedure 3: Flatten

```
Data: Auto $\mathcal{A}$
1  if $\mathcal{A}$ is a Prod then
2      Flatten each automata of $\mathcal{A}$
3      Product all flattened automata together
4  else // $\mathcal{A}$ is a Hier
5      foreach Node of $\mathcal{A}$ do
6          Flatten the sub-automaton of the Node
7          Break the frontier of Node
```

Procedure 4: BreakFrontier

```
Data: Node N
1  // N sub-automaton is assumed to be flat
2  add a unique new counter $\beta$ on all N incoming arrows
3  adapt constraints of N (multiply constants by $\beta$)
4  copy constraints of N in the parent automaton
5  match incoming arrows with intern arrows
6  match outgoing arrows with intern arrows
7  remove N and unreachable nodes
```

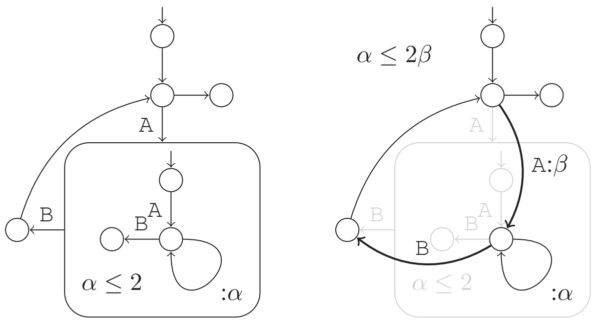

Fig. 11. Illustration of BreakFrontier

B. Flattening phase

Procedure 3 is able to transform a PPA into a flat automaton. It relies on the *BreakFrontier* procedure (at line 7) and the classical automata product (at line 3). Both algorithms only operate on flat automata, and this *Flatten* algorithm mainly serves this purpose. The recursive calls start by flattening the deepest automata, up to the top.

Procedure 4 is applicable on a specific *super-node* of a hierarchical automaton. The objective is to remove the context represented by the node while merging the sub-automaton into the parent automaton. For this purpose, the *node* has to be replaced by its sub-automaton and the constraints it carries must be adapted to maintain the context-sensitivity. This is done by introducing a new *counter* at line 2 and by the multiplication of constants of all constraints by this *counter* (see [5], [6]). Figure 11 illustrates the *BreakFrontier* algorithm applied on a super-node.

The product mentioned at line 3 of Procedure 3 refers to the classical automata product. The algorithm is adapted in order to handle the counters: in the result, counters are combined using the disjoint union (renaming may happen).

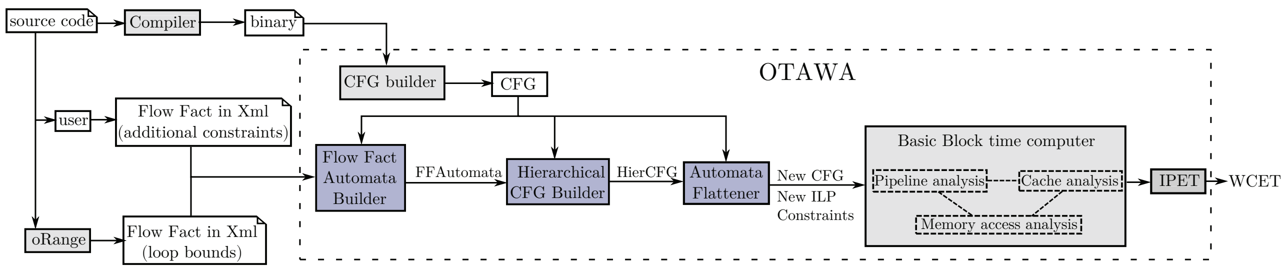

Fig. 12. Overview of our WCET analysis system with automata implementation

V. Experiments

A. Tool chain

In order to demonstrate the efficiency of our method, we implemented our automata as a plug-in into the WCET analysis toolkit OTAWA [7]. This academic tool dedicated to timing analysis supports multiple instruction sets such as *PowerPC*, *ARM*, *TriCore*, *Sparc* and several micro architectures.

Figure 12 presents our complete WCET analysis system, focusing on modules related to the automata described in this paper. The work flow begins with the source code to analyse and the associated binary file. On the left side, the academic tool oRange [8] is used on the source code to automatically generate a Flow Fact in Xml (FFX) file which contains contextual loop bounds. Using the same FFX format, the user can provide its own properties on the code such as exclusivity between conditions. On top of the figure, the binary and its debugging information are used by OTAWA to build the CFG of the program. This CFG and the FFX files are the central points of the analysis.

The *Flow Fact Automata Builder* module is responsible for the parsing of the FFX files. Using the edges of the CFG, it is able to generate an automaton that represents all the properties described in FFX. This PPA is fed to the *Hierarchical CFG Builder* that uses the *InjectionProduct* to inject the CFG into the PPA. The *Automata Flattener* is then fed with the Hierarchical CFG to obtain a flat CFG with unfolded/removed parts and additional ILP constraints. All known high level properties are now enclosed in this new CFG with its ILP constraints and it can be sent to the low level analysis, then to IPET for the WCET computation.

B. Existing benchmarks

We used an architecture model derived from ARM9 processor of OTAWA with a 1 kB instruction cache and a 16 kB data cache. The instruction cache used is a two-way associative cache with a Least Recently Used (LRU) replacement policy. In these experiments, we used programs from benchmarks of WCET Tool Challenge 2014 [9]. However we selected only few of them because for now, infeasible paths research is done manually, and we were not able to find obvious infeasible paths in the remaining of the benchmark suite.

Results of our experiments are presented in the Table I. The *bounded loops* column presents WCET computation with the only knowledge of maximum number of iterations of each loop to ensure basic finiteness of the program. The other columns present results from the use of automata to encode constraints into the CFG or the ILP system. In the *additional constraints* column, exclusivity constraints have been encoded as counters and constraints in a single state automaton and the resulting product with the CFG is the same as the initial CFG, with a richer ILP system. In the *unfolded* column, exclusivity constraints have been encoded as a two-state automaton and the resulting product with the CFG is an unfolded CFG. Both automata did capture the constraint correctly, which gives more than 10% precision gain. We could expect some low-level side-effect improvements for the unfolded versions, but there happens to be none in these benchmarks. WCETs are therefore identical for both automata versions.

TABLE I. WCET results with and without automata

	bounded loops	additional constraints		unfolded	
Program	WCET	WCET	Gain	WCET	Gain
runFlightPlan	1715	1534	10.6%	1534	10.6%
tcas-a	19880	17484	12.0%	17484	12.0%

TABLE II. WCET results for the overview example

	bounded loops	additional constraints		unfolded	
Program	WCET	WCET	Gain	WCET	Gain
excl_mod	419810	312900	25.5%	253556	39.6%

The program *runFlightPlan* comes from the benchmark *heli* which is a control software for a helicopter model, and *tcas-a*, derived from the benchmark *tcas*, is a traffic collision avoidance software. In these programs, we added constraints to express exclusivity between conditions in specific functions.

C. Overview example

The program presented in the overview (Section II) is a simplified version of the *excl_mod* program of Table II. This experiment shows a precision gain even between the WCET with additional constraints and the WCET with an unfolded version of the CFG, due to instruction cache effects. The program is composed of a simple loop containing two expensive blocks *A* and *B* that are mutually exclusive, followed by a common expensive block *C*. First, results show a precision improvement between the WCET with only basic finiteness and the WCET with additional constraints expressing the exclusivity. This constraint was added in the ILP system:

$$A\_occurences + B\_occurrences \leq Number\_of\_iteration$$

This global constraint ensures that at the end of the execution, we were not able to go through *A* or *B* more than the number of iterations of the loop. Without this constraint, the Worst-Case Execution Path (WCEP) was to go through *A* and *B* in

TABLE III. AT MOST K THEN TAKEN OVER n IF-THEN STATEMENTS.

K	feasible paths	program family
n	2^n	
$n-1$	2^n-1	dense_n
$n/2$	about $2^n/2$	half_n
1	$n+1$	sparse_n

every iteration of the loop. The 25.5% that we can see in the *gain* column comes from the removal of these infeasible paths.

The unfolded version shows a 14% additional gain which is mainly explained by the behaviour of the instruction cache and the associated memory access penalty. It has been mentioned that *A*, *B* and *C* were expensive blocks, but more precisely, they can each fill one of the two ways of the instruction cache, and with the LRU policy, in the sequence $A \to B \to C \to A$ *etc.* each block will be evicted from the cache right before it would be needed again.

If we look at the cache behaviour with the additional ILP constraint version, we can understand that since this constraint is global, and with an important memory access penalty, the WCEP found in reality is to go through *A*, *B* and *C* in half of the iterations and only in *C* for the other iterations. If we focus on the first half of the iterations, and as explained earlier, the sequence of blocks will result in the constant eviction of *A*, *B* and *C* from the instruction cache. In this situation, each instruction of these blocks will be tagged with *always-miss*. Moreover the other half of the iterations where instructions of the block *C* will be tagged with *always-hit* will not be considered by the static analyser due to the pessimism required by the WCET analysis. Therefore instructions of *C* will be tagged with *always-miss* as well.

In the unfolded version, the sequence $A \to B \to C \to A$ *etc.* has disappeared due to the path unfolding (see Figure 4). Therefore, the WCEP becomes $A \to C \to B \to C \to A$ *etc.* which makes each block *A* and *B* evict each other from the cache while *C* is not evicted any more from the cache. As a result, instructions of *C* are now tagged with *first-miss*, which participates in the improvement of the WCET precision.

D. Micro-architectural side-effect improvements

We experimented our plug-in on families of programs designed to explore several quantities and densities of infeasible paths. The families are sparse_n, half_n and dense_n where n is a number. Each of these programs contains the same function `main` which consists of a sequence of n `if-then` statements in a predictable `for` loop. Each loop body thus contains 2^n syntactic paths. The only path restriction that can be stated is a limit on the number of `then` taken during any iteration of the loop. Three variants of this limitation shown in Table III give birth to the three families.

Figure 13 presents computed WCET of the sparse_n program family with an increasing number of conditions (from 2 to 30) in the loop. The path restriction stated in Table III can be expressed as a PPA in order to include it in the WCET analysis process. Two types of PPAs are compared here but the liberty of the automata formalism allows to create other types of PPAs to express this specific property. The first option presented here is to include this property as a constraint in an iteration context (e.g. in Figure 2a). This constraint will

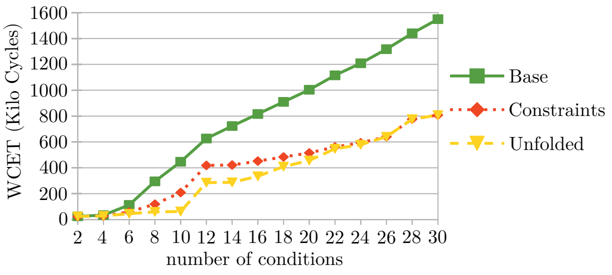

Fig. 13. Computed WCET depending on the number of `if-then` statements of the sparse_n program

be adapted during the flattening process and will result in a global ILP constraint depending on the number of iterations of the loop. Another option presented here is to express this property with a two-nodes automaton in an iteration context (e.g. in Figure 2b) which will result in an unfolding of the CFG and a duplication of some basic blocks.

The first noticeable fact is that the inclusion of any of the two PPAs described improves the WCET precision. Indeed, with no information about the `if-then` statements, the path selected by the static analyser for the safe WCET computation was going through every `then` branch in every iteration of the loop. Any of the two PPAs removes this infeasible path (and many others) which leads to this precision gain. Note that the basic blocks of the `if-then` statements are not negligeable and are part of the loop, which takes an important part in the WCET increase for the *Constraints* and *Unfolded* curves. Moreover none of the PPAs presented here forbids to go through a different `if-then` statement in each iteration of the loop, which will result in multiple cache penalties and will again increase the WCET.

The other point to consider is the delta between the unfolded version of the PPA and the constraint version. The WCEP for the constraint version becomes longer with every additional condition while the WCEP for the unfolded version is only composed of a few basic blocks in each iteration. Similar to the example presented in Section V-C, in the unfolded case and considering a well-suited cache size, instructions of the first and last block could be tagged with *first-miss* instead of *always-miss* which results in this WCET precision improvement.

E. Scalability

In order to study the scalability of the WCET computation time, each family of program has been analysed with a more important number of `if-then` statements. Results are presented in the Figure 14. *Base bounded loops* and *Auto bounded loops* curves correspond to the computation timings for basic finiteness of the program, using respectively the OTAWA FFX loader and our plug-in to inject loop information in the static analyser. *Constraints (all)* curve gathers timings for the three families of programs[4] presented in Table III where information on conditions were injected as constraints through automata in the analysis process. The last curve *Sparse Unfolded* shows results for the sparse_n family with unfolded PPA. The

[4]differences between these curves are invisible at this scale

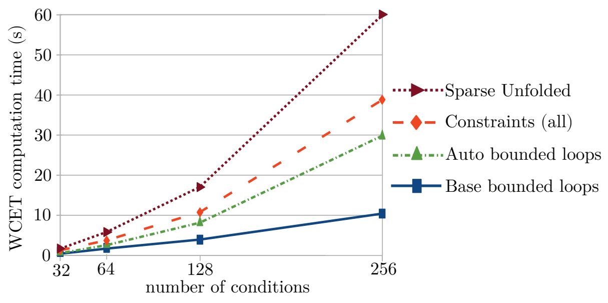

Fig. 14. Time needed to compute the WCET

unfolding of other families leads to an exponential explosion of the CFG. If we stick to basic finiteness, our tool does not provide WCET gain and gets slower than the built-in tool on large CFG. With richer infeasible paths, the estimation gain is consequent with respect to the extra analysis time. Unsurprisingly, unfolding costs more than playing with constraints.

VI. Discussion and Related Work

A. Automaton formalisms

Description of sequences using automaton comes initially from the language theory [10] and it has proved to be worthwhile in many domains. In particular, controller synthesis uses them in the same way that we do: for reducing the possible executions of a program. The closest work to ours is [4] where the notion of hierarchical automata is present, including the fact that an event can be used at several level. However, the presence of linear constraints on the occurrences of a transition brings our expressiveness beyond regular expressions (we can encode for example A^nB^n). Our proposal cannot be reduced to existing automata formalism. A close formalism is Timed Automata [11] which is well-suited for model checking (as in [12]). These automata embed free-running clocks that follow the real-time execution of the system. Our proposal is directed towards static analysis with IPET and focuses on number of executions of edges in complete finite traces and not on continuous timing properties.

B. WCET annotation languages

Feeding information to the WCET analyser in order to tighten (or simply enable) the estimation is not new. Each tool has one or several annotation languages to serve that purpose. In [1] Kirner et al. survey most of them and gives some classification criteria. If we apply these criteria to our formalism we obtain:

- A path-complete language. Able to describe all control flow path of arbitrary terminating programs.
- Information located out of the code and applicable to the binary CFG. The use of our proposal at the source code level would require an agreement with the user on a source-based CFG.

In the following, we compare our formalism to several annotation languages listed in [1] and also to a recent proposal [13].

Using automata modelling knowledge on the program was already the idea beneath the regular expressions of Park [14], used for path-based WCET. Our proposal can be seen as a transfer and adaptation of this idea to the field of IPET-based WCET analysis. The added features help the expression and integration of properties in an IPET-based context.

Engblom and Ermedahl propose an approach similar to ours in [5] where they consider function calls or loops as *scopes* of programs. They attach execution count variables on nodes representing basic blocks of the program and express flow information facts related to these variables in a specific scope. As they express these flow facts locally, they also address the problem of converting local constraints into IPET global constraints. However, their approach uses the CFG to directly attach counter variables, while we offer a degree of abstraction in the representation of flow facts information as well as a degree of liberty in the representation of the CFG: The arrows presented in our work are derived from edges of the CFG but they are a compact view of multiples edges where we can attach a unique counter, that will automatically be duplicated at the right place during the injection process. Moreover, our nodes are not directly related to basic blocks of the CFG, which gives us the liberty to encode properties in multiple ways, resulting for example in partial unfolding of the CFG (eg. Figure 13).

Our proposal contains few ingredients but expressive ones. It does not offer a specific structure for loop bounds, for branching incompatibility or for function scopes, but each of these notions can be encoded in our formalism. Having few but generic and powerful constructs is something we share with logic-based languages (e.g. [13], [15]). However the major difference is that we have been able to define (in Section IV) algorithms that perform integration of any automaton in a program CFG. Logic-based languages are so expressive that only a fragment of the logic can actually be used by a static analyser. The extent of the fragment and the way to take it into account in the analysis is often tool-dependent.

C. Information on data

Contrarily to languages like [13], [16] or [17], our formalism cannot express directly knowledge on data. Such knowledge must be fed to a static analyser able to turn it into knowledge on the program paths. From a WCET analyser point of view, the absence of information on data in our formalism is not a major issue and it will not require any extension of our proposal. The truth is that a WCET analyser can do little with information on data but to perform its own value analysis in order to prove the infeasibility of some paths. The values stored in the registers or in memory have almost no impact[5] on the timing analysis of a basic block.

D. Pertinence of the information

When a user or a static analyser provides paths properties not restricted to the loop bounds for a program, it is legitimate to ask whether this information will have any impact on the WCET estimation (other than slowing its computation). If the

[5]One can argue that the timing of some arithmetic operations can depend on their operand value on some architectures (e.g. multiplication on ARM9 can range from 3 to 6 cycles). However it is even more noticeable when these operations require a software emulation for which path properties can be expressed.

infeasible paths happen to be “short” paths, then the WCET estimation will probably be unchanged except in specific cases where it avoids imprecision on the low level analysis. Our proposal will give the best of itself in a context where the user/analyser has the capability to predict which information is likely to improve the WCET. A first approach to address this issue is the use of tools like the one proposed in [18] that identify C conditional statements that are syntactically unbalanced. Another approach is to create a feedback loop from the implicit Worst-Case Execution Path to the infeasible paths analysis. [19] and [20] automate such a feedback loop.

E. Sources of path properties

We can have several sources of path properties: the programmer, the compiler or dedicated static analysers. As described in Section V-A, our OTAWA plug-in automatically turns FFX files (generated by oRANGE or hand-written) into automata.

Describing an automaton in our formalism can be a pretty tedious task. For a higher user-friendliness we recommend the use of a language like the ones mentioned in Section VI-B and turn it automatically into an automaton. Once the ambiguities of that language have been resolved, the translation is pretty straightforward. Difficulties may arise when the initial language gives properties on the source code level but we believe it is a good thing that these difficulties arise at that specific moment rather than being melt with the difficulty of integrating the information.

Compilers are tightly linked with the infeasible paths issues since many infeasible paths can be removed by compiler optimisations. When activated they make our proposal less useful but they also make more difficult the transcription of source-level properties into binary-level properties. In [21], Li et al. allow loop bound annotation to be transformed according to the optimisation of the loops in the CFG.

We also mention works on automatic infeasible path detection and exploitation for specific WCET tools [22], [23] as potential source of path properties.

VII. CONCLUSION

We presented a formalism based on automata and linear constraints for modelling path properties, called Path Properties Automata (Section III). This formalism allows an expressive and compact representation of path restrictions relevant for the WCET analysis. We showed that it was suitable for effective use within an IPET-based WCET analyser (Section V).

The strength of our proposal is that integration of the infeasible path information does not require the modification of the WCET analysis itself. Instead we transform and decorate with linear constraint the control flow graph of the analyser using well defined operations on PPA. Algorithms for these operations are provided (Section IV). We showed that encoding infeasible paths using state-based acceptance might improve the low level accuracy in the WCET computation, with respect to linear constraints only.

The formalism we proposed opens several research tracks. The link between the hierarchical CFG and modular WCET analysis has to be explored. The capability to express path restrictions both explicitly and implicitly using either states or constraints offers an interesting way to drive the trade-off between cost and precision.

REFERENCES

[1] R. Kirner, J. Knoop, A. Prantl, M. Schordan, and A. Kadlec, “Beyond loop bounds: comparing annotation languages for worst-case execution time analysis,” *Software and System Modeling*, vol. 10, no. 3, pp. 411–437, 2011.

[2] G. Lesventes, *YAMS: Counter Automata Systems*. INRIA Research Report, Mar. 1987, no. 808.

[3] D. Harel, “Statecharts: A visual formalism for complex systems,” *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[4] F. Maraninchi and Y. Rémond, “Argos: an automaton-based synchronous language,” *Comput. Lang.*, vol. 27, no. 1/3, pp. 61–92, 2001.

[5] J. Engblom and A. Ermedahl, “Modeling complex flows for worst-case execution time analysis,” in *RTSS*, 2000.

[6] P. Raymond, “A general approach for expressing infeasibility in implicit path enumeration technique,” in *Proc. of EMSOFT*, New Dehli, 2014.

[7] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, “OTAWA: an Open Toolbox for Adaptive WCET Analysis,” in *Proc. of IFIP Workshop SEUS*, 2010.

[8] M. D. Michiel, A. Bonenfant, H. Cassé, and P. Sainrat, “Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation,” in *Proc. of RTCSA*, Taiwan, 2008.

[9] “WCET Tool Challenge,” http://www.mrtc.mdh.se/projects/WTC/.

[10] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to automata theory, languages, and computation (3. ed)*. Prendice Hall, 2006.

[11] R. Alur and D. L. Dill, “A theory of timed automata,” *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[12] A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen, “METAMOC: modular execution time analysis using model checking,” in *Proc. of WCET*, Brussels, Belgium, 2010.

[13] B. Lisper, “Principles for value annotation languages,” in *Proc. of WCET*, Ulm, Germany, 2014.

[14] C. Y. Park, “Predicting program execution times by analyzing static and dynamic program paths,” *Real-Time Systems*, vol. 5, no. 1, pp. 31–62, 1993.

[15] T. H. Kim, H. Bang, and S. D. Cha, “A systematic representation of path constraints for implicit path enumeration technique,” *Softw. Test., Verif. Reliab.*, vol. 20, no. 1, pp. 39–61, 2010.

[16] R. Chapman, A. Burns, and A. J. Wellings, “Combining static worst-case timing analysis and program proof,” *Real-Time Systems*, vol. 11, no. 2, pp. 145–171, 1996.

[17] J. Zwirchmayr, A. Bonenfant, M. de Michiel, H. Cassé, L. Kovacs, and J. Knoop, “FFX: A Portable WCET Annotation Language,” in *Proc. of RTNS*, 2012.

[18] J. Zwirchmayr, P. Sotin, A. Bonenfant, D. Claraz, and P. Cuenot, “Identifying relevant parameters to improve WCET analysis,” in *Proc. of WCET*, Ulm, Germany, 2014.

[19] J. Knoop, L. Kovács, and J. Zwirchmayr, “WCET squeezing: on-demand feasibility refinement for proven precise wcet-bounds,” in *Proc. of RTNS*, 2013.

[20] H. Bang, T. H. Kim, and S. D. Cha, “An iterative refinement framework for tighter worst-case execution time calculation,” in *Proc. of ISORC*, 2007.

[21] H. Li, I. Puaut, and E. Rohou, “Traceability of flow information: Reconciling compiler optimizations and WCET estimation,” in *Proc. of RTNS*, Versaille, France, 2014.

[22] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, “Efficient detection and exploitation of infeasible paths for software timing analysis,” in *Proc. of DAC*, San Francisco, USA, 2006.

[23] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, “Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution,” in *Proc. of RTSS*, Rio de Janeiro, Brazil, 2006.