Abstract—Computing the worst-case execution time (WCET) of real-time tasks is mandatory for the correct functioning of real-time systems. In component-based development (CBD) of real-time embedded systems, a task is typically constructed by composing appropriate software components together. Finding the WCET of a component-constructed task reduces to finding the WCETs of its constituent components and composing them in an appropriate fashion.

In this paper we describe a method based on constraint-logic programming (CLP) to produce WCET estimates of real-time tasks where the components’ WCETs are expressed parametrically. The experimental results show that the derived WCETs using our approach benefit from an increased tightness while the practical (time) complexity of the approach is manageable.

I. INTRODUCTION

Real-time systems are subject to scheduling and schedulability analysis to ensure they meet their timing requirements. A pre-requisite for this timeliness verification is knowledge about the worst-case execution time (WCET) of the tasks in the real-time system – which is the longest execution time that the task will ever exhibit on a particular hardware architecture. Techniques exist to derive the WCET of a given task using dynamic testing, static analysis, and hybrid measurement-based analysis. A comprehensive survey is available in [1].

Traditionally, the WCET of the task is derived by analysing/testing its code (and target hardware) which is typically written by the same developer. However recently, there has been a shift of focus to component-based development (CBD) of embedded software [2] to improve maintainability, reusability, and testability of real-time tasks, to name but a few. One impact of this on WCET analysis is that a real-time task is built from a set of components which are potentially shipped from different producers [3]. These components are “glued” together to form the real-time task of interest. In order to find the WCET of the task in this case, one can proceed in different ways depending on the nature of the software components that constitute this task.

When the source code of the components is available, the task is built by (appropriately) putting together this source code. In this case, traditional techniques to derive the WCET can be applied on the formed task. This method however is not preferred in CBD of real-time systems as a component can potentially be used as part of different (computer) tasks and the effort to estimate the WCET of the component is repeated during the WCET derivation of each of these tasks. This violates the reusability attribute of CBD of software.

Therefore, it is crucial that the WCETs of components are estimated once and for all, which has considerable benefits. On the one hand, if the component’s WCET estimation is made at the producer site, it can be shipped with the component and made available for the user of the component to employ in estimating the WCET of the task where this component is used. On the other hand, if the WCET of the component is performed at the user end, it will still make sense to do it only once and use it whenever the computation of the WCET of a task which uses this component is desired. In addition to saving analysis or/and testing effort, deriving the WCET of a task from its building components has the potential advantage of yielding tight values if the WCETs of the constituent components are context-sensitive to the usages or modes of operation of these components.

In order to make composability of component WCETs possible and useful, we need to address two issues. First, the WCET of a component must be expressed in a way that makes composition worthwhile. This means that the WCET of a component must be parametrised or predicated in terms of the possible usages of this component. For example, a component’s WCET expression of the type \( c_i = \text{predicates} \) or \( a_{i_1} > 0 \Rightarrow c_i = 10 \times a_{i_2} \) – where \( a_{i_1} \) and \( a_{i_2} \) are input parameters to the component – is suitable for compositional WCET analysis. Second, the composability must exploit the context sensitivity of the component WCETs and derive the WCET of the task in a safe and scalable manner.

The contributions of the paper are the following. First, we show that we can use constraint-logic programming (CLP) to obtain the WCET of a task composed of a number of components whose WCETs are expressed in a context-sensitive way. Second, we show how to augment WCET calculation with functional-behaviour constraints that capture the usages of the components at runtime – for better accuracy in estimating the WCET. Third, we show three levels of compositional analysis that differ in their time complexity and tightness of derived WCET. Fourth, we
conduct a thorough evaluation that shows the practicality of our approach.

The rest of the paper is organised as follows. In Section II we review related work. In Section III, we formulate our problem as a set of timing and functional-behaviour constraints to enable efficient WCET composability. In Section IV, we describe the timing-behaviour expressions, and in Section V, we describe the functional-behaviour expressions. In Section VI, we discuss the sources of tightness and how they relate to the timing and functional expressions. In Section VII, we discuss alternatives for performing the WCET calculation in the presence of timing and functional constraints. In Section VIII, we evaluate our approach using the prototype tool that we have implemented. Finally, Section IX contains insightful conclusions and pointers to future work.

II. RELATED WORK

Compositional timing analysis is motivated in [4] and mentioned in [5]. The works make no attempt to solve the composability problem. In [6], the effects of some hardware accelerators on an additive compositional WCET calculation are evaluated. The work does not attempt to perform compositional WCET analysis. In [7], the calculation of a composed overall WCET from the WCETs of partially analysed software components is presented. The focus of the work is the partial analysis of the cache behaviour of some software component and the integration of the partial results in an overall WCET estimate. In our work, we also consider composable WCET analysis, but with emphasis on parametrised usages instead of processor-behaviour analysis.

Parametric timing analysis is the subject of many works [8]–[13]. Each work discusses different methods for obtaining parametric timing expressions. With respect to our work, we are only interested in the format of the parametric expressions as they are input to our analysis. In general, a parametric expression is a conjunction of a number of predicated polynomials where the predicates are formed by linear expressions, and the polynomials have associated degrees. Most derived polynomials [8], [9], [11] have reportedly a degree of at most 3. In our work, we use parametric expressions like these ones – which are generated by state-of-the-art parametric WCET analysis.

Constraint-Logic Programming for WCET analysis is used in [14], [15] to perform static context-sensitive processor-behaviour analysis and measurement-based analysis. In our work, we use CLP at the component level.

III. PROBLEM FORMULATION

The objective of this paper is to show how one can estimate the WCET \( \text{wcet}_t \) of a task \( t \) composed of \( n \) software components \( C_i \) whose WCETs \( c_i \) are expressed in a context-sensitive way. The software components are connected together as dictated by the required functionality of task \( t \), and hence form a (connected) directed component graph \( (G = (V, E)) \) where \( V \) is the set of components and \( E \) is the set of edges between them. Each component \( C_i \) has also associated an execution count \( x_i \) which specifies the number of times \( C_i \) executes for a single run of task \( t \). In addition to this, an edge in the (directed) graph that links two components \( C_j \) and \( C_i \) has execution count \( x_{ji} \).

The problem is then to find the path of components in \( G \) with the longest-combined execution times of components \( C_i \). We shall formulate this problem as an implicit-path enumeration technique (IPET) [16] problem since it is the most natural way to model a graph-based problem with constraints as we shall see in this section. The WCET \( \text{wcet}_t \) of the task \( t \) is found by maximizing the sum in Formula (1).

\[
\text{wcet}_t = \sum_{i=1}^{n} x_i \cdot c_i
\]  

Each component \( C_i \) in graph \( G \) has its WCET \( c_i \) expressed in terms of its inputs \( A_i \) through some context-sensitive function \( g_i \) as shown in Formula (2).

\[
c_i = g_i(A_i)
\]  

For each component \( C_i \), a constraint is added to the model to state that the usage scenario of \( C_i \) is dictated by its immediate predecessor \( C_j \). For this, we need to introduce the notion of “when \( C_j \) and \( C_i \) execute in this order, the outputs of \( C_j \) become the inputs of \( C_i \)”. Hence, we define for each component \( C_i \) an output vector \( B_i \). The way we express inter-component functional behaviour is shown in Formula (3) where the term \( \text{ipred}(C_i) \) refers to the set of components which are immediate predecessors of \( C_i \), and the term \( c_{i/j} \) is the execution time of component \( C_i \) given its immediate predecessor \( C_j \) executes. In Formula (3), the execution of each component \( C_i \) is replicated in the number of its immediate predecessors \( C_j \). The component \( C_i \) acquires the execution times \( c_{i/j} \) which are defined over the outputs of \( C_j \) the immediate predecessor of \( C_i \) that shares the edge \( x_{ji} \) with \( C_i \).

\[
x_i = \sum x_{ji}, C_j \in \text{ipred}(C_i) \land c_i \in \{c_{i/j} \cdot C_j \in \text{ipred}(C_i)\} \land \forall j, C_j \in \text{ipred}(C_i) \cdot c_{i/j} = g_i(B_j)
\]  

Now that a component gets its inputs from its immediate predecessors in the component graph, we can define functional-usage scenarios for the individual components. Formula (4) states that if some component \( C_i \) executes, it produces output \( B_i \) given that it consumes input \( A_i \) according to some functional-behaviour function \( f_i \).

\[
x_i > 0 \iff B_i = f_i(A_i)
\]  

The objective function in Formula (1) is updated to reflect the conditional execution times \( c_{i/j} \) as shown in
on parametric WCET analysis (see Section II) where \( g \) is the timing-behaviour expression, \( pp \) means predicated polynomial, \( \text{linpred} \) is a linear predicate, and polynomial is an arbitrary degree polynomial expression over the inputs \( A_i \) of \( C_i \). The degree of the polynomial has a practical upper bound which normally corresponds to the deepest loop nest in the software component. Although loop nests in a program can grow to arbitrary sizes in theory; when the program is real-time, loop nests of small depth are typical, otherwise the program cannot produce the answer in “real-time”. The linear predicate \( \text{linpred} \) in Grammar (6) is constructed by combining linear polynomials \( \text{linpoly} \) via logical operators \( op \). The predicate \( \text{linpred} \) is set to true to obtain a non-predicated polynomial.

V. FUNCTIONAL-BEHAVIOUR EXPRESSIONS

Software components which are primarily developed for heavy reuse are typically shipped with well-defined usage contracts that specify the functional behaviour of the component. A usage contract can vary in its expressiveness from broadly stating whether the component will behave correctly or incorrectly given that its inputs are of some type, to accurately specifying values and ranges of inputs and outputs.

With respect to our work, the usage contract is a function that maps the inputs of the component to its outputs. In general, a component \( C_i \) has an associated input vector \( A_i \) with input space \( S(A_i) \), and has also an output vector \( B_j \) with output space \( S(B_j) \). A usage contract is a function that maps elements of \( S(A_i) \) to elements of \( S(B_j) \). An element-to-element mapping of \( S(A_i) \) to \( S(B_j) \) is usually not feasible, and hence, the mapping normally exists between subsets of \( S(A_i) \) and subsets of \( S(B_j) \). A more feasible and relaxed input-output contract expression is shown in Formula (7) which states that there is a usage contract that maps some inputs \( \{a_{ik_i} \in A_i \} \) of component \( C_i \) to some outputs \( \{b_{ik_2} \in B_j \} \) by respectively imposing constraints \( cs_{ik_1} \) and \( cs_{ik_2} \) on their respective domains \( \text{dom}(a_{ik_1}) \) and \( \text{dom}(b_{ik_2}) \).

\[
\begin{align*}
\exists a_{ik_1} \in A_i & \quad \land \quad \exists b_{ik_2} \in B_j \\
& \quad \land \quad cs_{ik_1}(\text{dom}(a_{ik_1})) = cs_{ik_2}(\text{dom}(b_{ik_2}))
\end{align*}
\]

(7)

Formula (7) implements a contract between one input variable \( a_{ik_1} \) and one output variable \( b_{ik_2} \). A usage contract will in general have more complex interactions between inputs and outputs e.g., a conjunction of constraints on input variables leading to a disjunction of constraints on output variables. Such complex expressions can be generated by conjuncting and disjuncting instances of Formula (7).

VI. CONTEXT SENSITIVITY

In order to reason about the tightness of the WCET estimation provided by our approach, we need to have a deeper...
behaviour expression\[\hat{c}\] returns an execution time analysis which are all derived from the model represented by

For this purpose, we discuss three types of compositional insight in the ways by which our method is context sensitive.

A. Sensitive Compositional Analysis

Sensitive compositional analysis is obtained when all the constraints listed in Table I are included in the model. In this type of analysis, the returned execution time \(c_i\) of component \(C_i\) is sensitive to which immediate predecessor has been executed before it in the returned longest path.

B. Semi-Sensitive Compositional Analysis

In semi-sensitive compositional analysis, we calculate \(\text{wcet}_t\) of the task \(t\) using constant values of the WCETs \(c_i\) of the components \(C_i\). Computing a constant \(c_i\) reduces to solving the constraint-optimization problem (COP) where \(c_i\) is the value to be maximized. Table II shows the COP used to find the maximum value \(c_0\) of the execution time \(c_0\) of some component \(C_0\). In order to find \(\hat{c}_0\), we restrict our problem to component \(C_0\) and its immediate predecessors only (row 1) as they are the only components that affect its inputs \(A_0\). The functional-behaviour expression \(f_0\) together with the outputs \(B_0\) are dropped since the effect of executing \(C_0\) on its successors is not of concern (row 2). The timing-behaviour expression \(g_j\) of each immediate predecessor \(C_j\) is dropped (row 3) as they are of no concern in finding \(c_0\). We then impose the necessary structural constraints (rows 4-5), functional constraints (rows 6-8), and timing constraints (rows 9-10).

The value \(\hat{c}_i\) is calculated for every component \(C_i\), and together with the necessary structural constraints on the execution counts \(x_i\), the value \(\text{wcet}_t\) is computed.

<table>
<thead>
<tr>
<th></th>
<th>FORMULATING THE EXPRESSION (c_0) OF COMPONENT (C_0) AS A COP WHERE (c_0) IS AWARE OF THE USAGES OF THE OUTPUTS OF THE IMMEDIATE PREDECESSORS OF (C_0).</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(G = (V, E), V = {C_0} \cup \text{spred}(C_0),</td>
</tr>
<tr>
<td>2</td>
<td>(C_0 = (A_0, \emptyset, \emptyset, g_0))</td>
</tr>
<tr>
<td>3</td>
<td>(\forall j, C_j \in \text{spred}(C_0) \bullet C_j = (A_j, B_j, f_j, \emptyset))</td>
</tr>
<tr>
<td>4</td>
<td>(x_0 = \sum x_{j0}, C_j \in \text{spred}(C_0))</td>
</tr>
<tr>
<td>5</td>
<td>(x_0 = 1)</td>
</tr>
<tr>
<td>6</td>
<td>(\forall i \leq n, \forall a_{ik} \in A_i \bullet a_{ik}^{\text{in}} \leq a_{ik} \leq a_{ik}^{\text{out}})</td>
</tr>
<tr>
<td>7</td>
<td>(\forall j, C_j \in \text{spred}(C_0), \forall b_{jk} \in B_j \bullet b_{jk}^{\text{in}} \leq b_{jk} \leq b_{jk}^{\text{out}})</td>
</tr>
<tr>
<td>8</td>
<td>(\forall j, C_j \in \text{spred}(C_0) \bullet x_{j0} &gt; 0 \iff b_j = f_j(A_j))</td>
</tr>
<tr>
<td>9</td>
<td>(\forall j, C_j \in \text{spred}(C_0) \bullet c_{0/j} = g_0(B_j))</td>
</tr>
<tr>
<td>10</td>
<td>(\forall j, C_j \in \text{spred}(C_0) \bullet c_0 = \max(c_{0/j}))</td>
</tr>
</tbody>
</table>

The main difference between sensitive compositional analysis and semi-sensitive compositional analysis is that in the latter, the execution time \(c_i\) of a component \(C_i\) with more than one immediate predecessor is not sensitive to the immediate predecessor that executes before \(C_i\) in the returned path.

C. Non-Sensitive Compositional Analysis

In non-sensitive compositional analysis, the execution time \(c_0\) of some component \(C_0\) is completely oblivious to the usages imposed by the immediate predecessors \(C_j\) of component \(C_0\) as shown in Table III. The value \(c_0\) is computed by solving the COP shown in Table III. The estimation of \(\text{wcet}_t\) is computed by accounting for the values \(c_i\) and the structural constraints on \(x_i\).

The main difference between non-sensitive compositional analysis and the two previous versions, is that the execution time \(c_i\) is not aware of the immediate predecessors and is computed based on a non usage-constrained expression \(g_i\).

D. Sources of Context Sensitivity

We can finally list the sources of context sensitivity in compositional analysis. The amount of tightness gained by each source of context sensitivity depends on the timing expressions \(g_i\) and functional expressions \(f_i\).

Parametric Sensitivity. The execution time \(c_i\) of each component \(C_i\) is expressed parametrically in terms of the input \(A_i\) through expression \(g_i\). Sensitive, semi-sensitive, and non-sensitive compositional analyses have parametric sensitivity. The amount of tightness gained by parametric sensitivity depends on the context sensitivity of expression \(g_i\).

Usage Sensitivity. The input \(A_i\) is constrained by the outputs \(B_j\) of the immediate predecessors \(C_j\) which are constrained by their turn via the expressions \(f_j\). Sensitive and semi-sensitive compositional analyses have usage sensitivity. The amount of tightness gained by usage sensitivity is proportional to the difference between the maximum execution times \(\hat{c}_i\) and the execution times \(\max(c_{i/j})\) given immediate predecessors \(C_j\). For example, if \((\hat{c}_i \gg \max(c_{i/j}))\) is true for every component \(C_i\), usage sensitivity will be responsible for a great part of the tightness in the estimation of \(\text{wcet}_t\).

Path Sensitivity. The execution time \(c_i\) is sensitive to exactly which immediate predecessor has executed, or to the execution counts of the immediate predecessors in the
case of loops. Only sensitive compositional analysis has path sensitivity. The amount of tightness gained in this case depends on the nature of the expressions $g_i$ and $f_i$ and how the expression $g_i$ of some $C_i$ is affected by the expressions $f_j$ of its predecessors.

VII. Calculation

Finding the WCET $\text{wcet}_t$ of the task $t$ reduces to finding the path with the longest execution time in the component graph $G$. At this analysis level, we can apply calculation techniques from the WCET analysis literature which are namely tree-based calculation, path-based calculation, and IPET-based calculation. These calculation techniques are conventionally used at the basic block level, in this work, we use them at the component level.

A tree-based approach – which is based on an abstract-syntax tree suitable for a quick calculation of the WCET but inadequate for propagating constraints over the nodes of the tree. A path-based approach is feasible when the number of paths in the component graph is manageable. In this case, all paths are enumerated, the functional-behaviour expressions are propagated through all the components of the path, and the timing-behaviour expressions evaluated accordingly. The applicability of a path-based method depends on the complexity of the component graph. In our work, we target arbitrary component graphs (cyclic and acyclic) and so a path-based approach is ruled out.

Consequently, the most efficient way to perform the compositional analysis is through the IPET technique. The notion of global propagation of functional behaviour must be present at all times if true tightness is sought, and IPET just provides enough expressiveness to model a global view of the interaction between components.

The general problem that we want to solve as IPET is the one formulated in Section III and which has the constraints in Table I. The IPET approach has almost always been synonymous to using integer-linear programming (ILP) although other techniques such as CLP are also able to provide a global view of the constraints on the graph and “implicitly enumerate” the paths.

Standard ILP cannot be used to completely model our problem because ILP requires that the execution times $c_i$ be constant. Furthermore, our model contains logical equivalences which are disjunctions by definition, and so adhoc adaptations of the ILP formulation are needed which were shown [15] to yield IPET problems that require exponential time to solve. On the other hand, CLP is able to handle arbitrarily-complex constraint expressions which in our case are the timing-behaviour expressions $g_i$ and the functional-behaviour expressions $f_i$.

In order to calculate $\text{wcet}_t$ using sensitive compositional analysis, CLP will be used since ILP cannot handle all the constraints in Table I.

In order to calculate $\text{wcet}_t$ using semi-sensitive (respectively non-sensitive) compositional analysis, CLP will be used to compute the values $c_i$ according to Table II (respectively Table III) and then the resulting constant values $c_i$ are used to compute $\text{wcet}_t$. The constant values $c_i$ can be combined in a tree-based approach or formulated as an ILP problem which in this case will be a network-flow problem by which solving its non-integer relaxation we obtain a (cheap) integral solution [17]. Since the difference in computational cost between a tree-based approach and network-flow ILP approach is marginal in our case, we opt for the ILP version as it is more convenient because of ease of implementation.

VIII. Evaluation

The input to our method is a directed connected graph $G$ of $n$ components $C_i$ labelled with their timing-behaviour expressions $g_i$ and functional-behaviour expressions $f_i$. The objective is to find $\text{wcet}_t$ which is the WCET of the task $t$ represented by $G$.

A. Experimental Setup

The components $C_i$ and their associated expressions $g_i$ and $f_i$ are generated automatically and randomly. The reason behind this is that all that matters in evaluating our approach is the structure and size of the graph $G$, and the degree and complexity of the expressions $g_i$ and $f_i$. This allows a more thorough and stronger evaluation of the approach as we are not constrained by using a limited set of benchmark programs.

Each experiment is conducted like the following. A randomly-structured graph $G$ of $n$ components is generated. For each component, an input vector $A_i$ of variables $a_{ik}$ and an output vector $B_i$ of variables $b_{ik}$ are defined. All vectors $A_i$ and $B_i$ have the same size $S$ and each element of each vector has the same domain $D$ without loss of generality.

$$\forall a_{ik} \in A_i, \forall b_{ik} \in B_i, 
\begin{align*}
m_1 \leq a_{ik} & \leq m_2 \Rightarrow m_3 \leq b_{ik2} \leq m_4 \\
\land \quad a_{ik}^{low} & \leq a_{ik} \leq a_{ik}^{up} \\
\land \quad b_{ik}^{low} & \leq b_{ik} \leq b_{ik2} \leq m_4 \leq b_{ik2}^{up}
\end{align*}$$

In addition to this, a timing expression $g_i$ of size $s$ and polynomial degree $d$, and which satisfies Grammar (6) is generated for every component $C_i$. The size $s$ of a timing expression $g_i$ is the number of conjuncts that correspond to the second rule in Grammar (6) i.e., the number of predicated polynomial expressions in $g_i$. The polynomial degree $d$ of the timing expression $g_i$ is the degree of the polynomial with the largest degree in $g_i$. In summary, $g_i$ is a conjunction of $s$ predicated polynomials of degrees in $[1..d]$ where at least one polynomial has degree $d$.

The functional expression $f_i$ is generated by conjuncting and/or disjuncting instances of Formula (7). In this evaluation, each function $f_i$ has the format shown in Formula (8).
The values \{m_1, m_2, m_3, m_4\} that constrain the variables in
the vectors \(A_i\) and \(B_i\) are randomly generated.

For sensitive compositional analysis, the problem is
completely expressed in ECLiPSe [18] the CLP language. The
hybrid finite domain and real-number interval constraint
solver ic [18] is used to express the constraints over the
variables in the problem. The returned solution is guaranteed
to be safe since complete search with branch-and-bound is
used to maximize the objective function in Formula (1). For
semi-sensitive and non-sensitive compositional analysis, the
execution times \(c_i\) are obtained by formulating and solving
them in CLP, then they are passed to lpsolve [19] the ILP
solver for the calculation of \(wcet_i\).

All experiments are conducted on an 32-bit x86 CPU
running at 2.8GHz and 4GB memory, on Ubuntu 9.10, using
ECLiPSe version 6.x, and lpsolve version 5.5.x.

B. Tightness

Our approach achieves tightness by using both the para-
metric timing expressions \(g_i\) and functional expressions \(f_j\).
To show the tightness provided by our method we evaluate
the three types of compositional analysis namely sensitive,
semi-sensitive, and non-sensitive analyses.

We have created a number of component graphs \(G\) with
sizes \(|V|\) between 1 and 1000, containing loops between 1
and 10, and each component has parents between 1 and 5.
For each generated \(G\), we compute the \(wcet_i\) using the three
types of analysis. The size of the expressions \(g_i\) is between
1 and 5, while their degree is at most 3.

The results are shown in Figure 1. As can be seen,
the best tightness is obtained using sensitive, then semi-
sensitive, then non-sensitive compositional analysis – which
is expected. On average – according to Figure 1, using
the sensitive analysis to compute the \(wcet_i\) gives 97.2%
tightness when compared to non-sensitive analysis while it
gives only 13.7% tightness when compared to semi-sensitive
analysis. Semi-sensitive analysis gives a very good tightness
of 73.0% on average compared to non-sensitive analysis.

According to the results in Figure 1, the tightness loss
between using sensitive analysis and semi-sensitive is not
substantial when compared to the considerable tightness
achieved in going from non-sensitive analysis to semi-
sensitive analysis. This suggests that path sensitivity is not
as important as usage sensitivity (Section VI-D) in providing
tightness. The results of Figure 1 might of course be biased
on the automatic model generation that we use. In the
general case, the tightness gained from usage and path
sensitivity might vary; all that Figure 1 shows is that there
is an incremental tightness gain in going from non-sensitive
to semi-sensitive to sensitive compositional analysis.

C. Scalability

In Section VIII-A, we have listed the parameters: \(n\) which
determines the size of the problem, \(s\) which determines the
number of predicated polynomials added by every compo-
ment, \(d\) which determines the non-linearity degree of the
problem, and \(S\) which is the size of vectors \(A_i\) and \(B_i\)
and the domain \(D\) of their elements. These are the variables
that affect the hardness of the COP that the compositional
analysis attempts to solve.

The hardness of performing semi-sensitive and non-
sensitive compositional analysis is linear in terms of the
variable \(n\). The effort in this case is spent computing
the execution time \(c_i\) of each component by solving its
associated COP as shown in Tables II and III. Solving the
COP of each \(c_i\) – which is NP-complete in the general case
– is repeated \(n\) times and hence the linear relationship. The
time required to solve the network-flow ILP is considered
constant and is very scalable. The hardness of performing
the sensitive compositional analysis is not guaranteed to be
linear in terms of \(n\).

The hardness of performing all types of compositional
analysis increases in terms of \(s\). Each of the \(s\) predicated
polynomials associated with component \(C_i\) creates a new
partition of the input space of the expression \(g_i\) in terms
of a set of constraints which must be propagated to the
constraints imposed by the expressions \(f_j\) of the immediate
predecessors \(C_j\). This increases the effort of the constraint-
satisfaction process.

Non-linear polynomials are harder to manipulate than
linear ones. When the constraint-satisfaction search is per-
formed over a term \((a \ast b)\) for example, the problem is
normally solved on \(a\) and \(b\) is suspended (or delayed) until
\(a\) is instantiated. Depending on the capability of the solver,
this delay-and-wait process may or not take a long time.

The size \(S\) of vectors \(A_i\) and \(B_i\) affect compositional
analysis considerably as the size of the search space in this
case is \((2 \ast n \ast S^{|D|})\) i.e., each of the \(n\) components has 2
vectors \(A_i\) and \(B_i\), and each vector contains \(S\) variables of
domain $D$ each.

We can define the complexity of our approach as some function $h$ whose value increases in terms of a vector $v$ of parameters $n, s, d, S, D$, and $|D|$. One way to evaluate the complexity of $h$ is to maintain constant all parameters in $v$ but one which is varied in order to observe its effect on the growth of $h$. This in our case will not yield meaningful results as the effects of the parameters in $v$ on $h$ are not independent. For example, it could be shown by the experiments that both $d$ and $s$ cause a linear growth of $h$ when increased individually, but this does not show the growth of $h$ when they both increase at the same time.

In order to give a more realistic complexity evaluation of our approach, we proceed like the following. First, we define domains for the parameters in vector $v$: $n \in [1..1000]$, $s \in [1..10]$, $d \in [1..5]$, $S \in [1..10]$, and $|D| \in [1..10^{10}]$.

The rationale behind choosing the previous values is the following. Working with component-based real-time tasks provided by our industrial partners reveals that $n$ is normally in the range $[1..30]$, by using the range $[1..1000]$ we show that we can handle much larger systems of components. From the literature on parametric WCET analysis: the degree of the polynomials is normally less than 3, we use polynomial degrees up to 5; the WCET expressions are defined over less than 10 variables, we use $S = 10$. We use a bigger domain than the integer domain ($|D|$). We limit our evaluation to parametric WCET expressions of size $s = 10$.

Second, we perform scalability evaluation according to Table IV. We evaluate the complexity function $h$ of the three types of analysis in terms of the vector $v = (n, s, d, S, D)$ for which we pick sensible values as shown in column 2 of Table IV. In order to obtain the next value of $v$, we increase the value of one of its parameters. In the general case, the complexity value $h(v)$ increases when at least one parameter of $v$ increases which can be formalised as shown in Formula (9). The tuples $v$ in Table IV have increasing values from top to bottom. We associate each tuple $v$ with an x-coordinate value (column 1) that we use together with the values $h(v)$ (y-coordinates) to plot the curve of $h$. The values $h(v)$ corresponding to sensitive (respectively non/semi sensitive) compositional analysis in column 3 (respectively column 4). The complexity for both non-sensitive and semi-sensitive analysis is very similar to a precision of $10^{-3}$ and hence both analyses share the same column.

\[
\begin{align*}
|v_i| &= |v_j| \\
&\wedge h(v_i) \leq h(v_j) \iff v_i \leq v_j \\
&\wedge v_i \leq v_j \iff \forall k \leq |v_i| \cdot v_i[k] \leq v_j[k]
\end{align*}
\tag{9}
\]

In our evaluation, the graphs $G$ are randomly generated which could lead to a scenario where a 100-component graph is easier to solve than a 50-component graph because the predicates on its polynomials are easier to resolve for instance. This example scenario could invalidate our evaluation results as it does not obey Formula (9). To overcome this problem, for each row of Table IV, a random problem is generated and solved 100 times using the value $v$ in that particular row, then the average solution time is taken and recorded in columns 3 and 4.

The complexity function $h$ for sensitive analysis is shown in Figure 2, and the one for semi/non sensitive analysis is shown in Figure 3.

For sensitive compositional analysis, we observe an exponential growth of the function $h$ which is not very surprising given the NP-completeness of constraint satisfaction and optimization. However, the most complex problem is solved in less than two minutes which is very promising.

Using non-sensitive and semi-sensitive compositional analysis, the most complex problem is solved within a tenth of a second. The growth of the complexity function is exponential as well; the reason being that the hardness of the COPs to solve within each component increases with increasing values of $v$, nevertheless, the overall growth is linear given the negligible time needed to solve the network-
flow ILP.

IX. CONCLUSION

We have presented and implemented a solution to compositional WCET analysis of component-based real-time tasks. We have shown that the composition of parametric WCET expressions augmented with functional-behaviour constraints give necessarily tight WCET estimates for the tasks. We have also shown that using our technique we can analyse tasks characterised by large systems of components which are interconnected through complex functional-behaviour expressions. Finally, we have shown that compositional analysis can be performed within manageable time.

Our next target is to derive better search heuristics for the constraint-programming solution to reduce the practical time complexity of compositional analysis. We intend to bring floating-point variables and constraints into the picture. We also aim to apply the technique on actual code provided by our industrial partners.

ACKNOWLEDGEMENTS

This work is supported by the Swedish Foundation for Strategic Research (SSF) through the Research Centre for Predictable Embedded Software Systems (PROGRESS).

REFERENCES


