Path-Sensitive Resource Analysis Compliant with Assertions

Duc-Hiep Chu
National University of Singapore
hiepcd@comp.nus.edu.sg

Joxan Jaffar
National University of Singapore
joxan@comp.nus.edu.sg

Abstract

We consider the problem of bounding the worst-case resource usage of programs, where assertions about valid program executions may be enforced at selected program points. It is folklore that to be precise, path-sensitivity (up to loops) is needed. This entails unrolling loops in the manner of symbolic simulation. To be tractable, however, the treatment of the individual loop iterations must be greedy in the sense once analysis is finished on one iteration, we cannot backtrack to change it. We show that under these conditions, enforcing assertions produces *unsound* results. The fundamental reason is that complying with assertions requires the analysis to be fully sensitive (also with loops) wrt. the assertion variables.

We then present an algorithm where the treatment of each loop is separated in two phases. The first phase uses a greedy strategy in unrolling the loop. This phase explores what is conceptually a symbolic execution tree, which is of enormous size, while eliminates infeasible paths and dominated paths that guaranteed not to contribute to the worst case bound. A compact representation is produced at the end of this phase. Finally, the second phase attacks the remaining problem, to determine the worst-case path in the simplified tree, excluding all paths that violate the assertions from bound calculation. Scalability, in both phases, is achieved via an adaptation of a dynamic programming algorithm.

1. Introduction

Programs use limited physical resources. Thus determining an upper bound on resource usage by a program is often a critical need. Ideally, it should be possible for an experienced programmer to extrapolate from the source code of a well-written program to its *asymptotic* worst-case behavior.

However, "*concrete* worst-case bounds are particularly necessary in the development of embedded systems and hard real-time systems." [14]. A designer of a system wants hardware that is *just good enough* to safely execute a given program, in time. As a result, precision is the key requirement in resource analysis of the program. Now embedded programs are often written in favor of performance over simplicity or maintainability. This makes many analytical techniques[1] [11, 14, 10] less applicable.

Fortunately, there are two important redeeming factors. First, embedded programs are often of small to medium size (from dozens to a few thousand lines of code) and symbolically executing them is guaranteed to terminate. Second, programmers are willing to spend time and effort to help the analyzer in order to achieve more accurate bounds. In many cases, often such manually given information, in form of what we shall call *assertions*, is essential.

The primary target of this paper is Worst-Case Execution Time (WCET) analysis. The result is directly extendable to analysis of other *cumulative* resource such as power. Extension towards analyses of non-cumulative resources, e.g., memory and bandwidth, is left as our future work. Furthermore, here we only deal with the high level aspects of the analysis. Architecture modeling, when applicable, is considered as a separate issue and is out of scope of the paper. In other words, we only address the *path analysis* problem. Though our path analysis is supposed to work at the level the control flow graphs (CFG), for better comprehension, all the examples we present are at the source code level.

The Need for Path-Sensitivity

```
t = i = 0;
while (i < 10) {
   if (i mod 3 == 0)
      { j *= j; t += 30; }
   else
      { j++; t += 1; }
   i++;
}
```

Figure 1: The Need for Path Sensitivity

Precise path analysis essentially arises from *path-sensitivity*, and this in turn essentially arises from the ability to disregard *infeasible paths*. But how do we deal with the subsequent explosion in the search space? In fact, due to loops, fully path-sensitive algorithms cannot scale. In practice, abstract reasoning with "path-merging" is used, e.g., [19, 8, 12, 13].

The example in Fig. 1 concerns Worst-Case Execution Time (WCET) analysis, or simply timing analysis. The special variable t captures the timing. The program iterates through a loop, using the counter i. In the iteration such that `(i mod 3 == 0)`, a multiplication is performed, thus requires 30 cycles to finish. Otherwise, an addition is performed, which takes only 1 cycle to finish. The main challenge is to discover that multiplications are in fact performed three times less often than additions. In general, such discoveries are very hard.

An easy solution is to perform *loop unrolling*. In Fig. 1, we start with $(t = 0, i = 0)$. In the first iteration, we detect that the **else** branch is infeasible. At the end of this iteration, we have $(t = 30, i = 1)$ (since j does not affect the control flow, we just ignore information about j). In the second iteration, as $(i = 1)$ we detect that the **then** branch is infeasible; from the other branch, we then obtain $(t = 31, i = 2)$. This process continues until $(i = 10)$, when we exit the loop (having discovered that multiplication is executed exactly 4 times, and the exact WCET of 126).

Clearly a direct implementation of loop unrolling cannot scale. Recently, [3] developed a fully automated symbolic simulation

[1] These are more general by the use of *parametric* bounds, and they discover a closed worst-case formula. But they are not generally used for concrete analyses.

EMSOFT'13, October, 2013, Montreal, Canada.
978-1-4799-1443-2/13/$31.00

algorithm which is fully path sensitive wrt. loop-free program fragments. For loops, the algorithm performs loop unrolling while employing judicious use of path-merging only at the end of each loop iteration, thus useful information is systematically propagated across loop iterations and between different loops. As already pointed out in [3], loop unrolling is almost *inevitable* in order to capture precisely infeasible path information and complicated loop patterns such as: non-rectangular, amortized, down-sampling, and non-existent of closed form. The main contribution of that work is an adaptation of a dynamic programming algorithm, which employs the concept of summarization with interpolant in determining reuse so that loop unrolling can be performed efficiently.

Importantly, loop unrolling now [3] is observed to have *super-linear* behavior for the set of WCET benchmark programs with complicated loops. Informally, this means that the size of the symbolic execution tree is linear, even for nested loop programs of polynomial complexity. The key feature that allows scalability of loop unrolling is not just that a single loop iteration is summarized, and then subsequent loop iterations are analyzed using this summarization. Rather, a *sequence* of consecutive loop iterations can be summarized and reused. This is crucial for when it matters most: when loops are nested.

The Need for User Assertions

It is generally accepted in the domain of resource analysis that often programs do not contain enough information for program path analysis. The reason is that programs typically accept inputs from the environment, and behave differently for different inputs. It is just too hard, if not impossible, to automatically extract all such information for the analyzer to exploit.

```
t = 0;
for (i = 0; i < 100; i++) {
   if (A[i] != 0) {
      /* heavy computation */
      t += 1000;
   } else { t += 1; }
}
```

Figure 2: Assertions are Essential

See Fig. 2. Each non-zero element of an array A triggers a heavy computation. From the program code, we can only infer that the number of such heavy computations performed is bounded by 100. In designing the program, however, the programmer might have additional knowledge regarding the sparsity of the array A, e.g., no more than 10 percent of A's elements are non-zero. We refer to such user knowledge as *user assertions*[2]. The ability to make use of such assertions is crucial for tightening the worst case bound.

In general, a framework — by accommodating assertions — will allow different path analysis techniques to be combined easily. As path analysis is an extremely hard problem, we do not expect to have a technique that outperforms all the others in all realistic programs. Under a framework which accommodates *assertions*, any *customized* path analysis technique can encode its findings in the (allowed) form of assertions, and simply let the aggregation framework exploit them in yielding tighter worst-case bounds. Two widely used commercial products for timing analysis are [1, 2], both accommodate assertions, giving evidence to their practical importance.

We comment here that we do not consider the *proof* of any given assertion. Proving the validity of a given assertion is simply considered as an orthogonal issue.

[2] Strictly speaking, they are user *assumptions* about the program and the analyzer simply takes them in as *asserted* knowledge.

Path-Sensitivity and Assertions Don't Mix

We have argued that we need both path sensitivity (up to loops) and assertions in order to have precision. In this paper, we propose a framework for path analysis in which *precise* context propagation is achieved by loop unrolling. In addition, we instrument the program with frequency variables, each attached to a basic block. Each frequency variable is initialized to 0 at the beginning of the program and is incremented by 1 whenever the corresponding basic block is executed. Importantly, our framework accommodates the use of assertions on frequency variables so that user information can be explicitly exploited. In other words, the framework not only attempts path-sensitivity via loop unrolling as in [3], but also makes use of assertions to *block* more paths, i.e., disregard paths which violate some assertion. Ultimately, we can tighten the worst-case bound.

Now because assertions, when they are used, are typically of high importance, we require our framework to be *faithful* to assertions. That is, all paths violating some given assertion are guaranteed to be excluded from the bound calculation process. This requires the framework to be *fully path-sensitive* wrt. the given assertions. However, to make loop unrolling scalable, a form of greedy treatment with path merging is usually employed. In other words, the analysis of a loop iteration must be finished before we go to the next iteration. Also, the analysis should produce only one single continuation context[3]; this context will be used for analysis of subsequent iterations or subsequent code fragments. It is here that we have a major conflict: unrolling while ensuring that we recognize *blocked* paths that arise because of assertions.

```
c = 0, i = 0, t = 0;
while (i < 9) {
   if (*) {B1: c++; t += 10; }
   else {
      if (i == 1) {B2: t += 5; }
      else {B3: t += 1; }
   }
   i++;
   assert(c <= 4);
}
```

Figure 3: Complying with Assertions in Loop Unrolling is Hard

See Fig. 3 where the special variable t captures timing, and "*" is a condition which cannot be automatically reasoned about, e.g., a call to an external function `prime(i)`. We also instrument the program with the frequency variable c which is incremented each time **B1** is executed. The assertion `assert(c <= 4)` constrains that **B1** can be executed at most 4 times.

We now exemplify loop unrolling. In the first iteration, we consider the first **then** branch, notice t is incremented by 10 so that we get $(t = 10, c = 1, i = 1)$ at the end. Considering however the **else** branch, we then detect that the **then** branch of nested **if-statement** is infeasible, thus we finally get $(t = 1, c = 0, i = 1)$. Performing path-merging to abstract these two formulas, we conclude that this iteration consumes up to 10 cycles ($t = 10$) and we continue to the next iteration with the context $(t = 10, c = 0 \vee c = 1, i = 1)$[4] (the longest execution time is kept). Note that by path-merging, we no longer have the precise information about c, i.e., we say this merge is *destructive* [22].

Now let us consider two mechanisms for making use of assertions to block invalid paths. They correspond to *must* semantics and *may* semantics, respectively.

[3] We can generalize this to a fixed number of continuation contexts.

[4] To be practical, one must employ some abstract domain for such merge. In our implementation, we use the polyhedral domain.

Now the first option is to block path with context which *must* violate the assertion. In other words, a path is blocked when its context conjoined with the assertion is unsatisfiable. Return to the example, if we perform merging as above, at the beginning of the fifth iteration, the context of c is $(c = 0 \vee c = 1 \vee c = 2 \vee c = 3 \vee c = 4)$. Executing **B1** gives us the context $(c = 1 \vee c = 2 \vee c = 3 \vee c = 4 \vee c = 5)$. Conjoining this context with the assertion $c \leq 4$ is still satisfiable. Therefore, we cannot make use of the assertion to block any path in the fifth iteration or in any subsequent iteration. We end up having the worst case timing of 90 cycles: each iteration consumes 10 cycles. Consequently, however, the analysis is not faithful to the provided assertion.

A naive alternative is to block path with context which *may* violate the assertion. In other words, a path is blocked when its context conjoined with the negation of the assertion is satisfiable. In the first four iterations, the execution of block **B1** is possible. As before, at the beginning of the fifth iteration, the context of c is $(c = 0 \vee c = 1 \vee c = 2 \vee c = 3 \vee c = 4)$. Executing **B1** gives us the context $(c = 1 \vee c = 2 \vee c = 3 \vee c = 4 \vee c = 5)$. Conjoining this context with the negation of the assertion, namely $c > 4$, is obviously satisfiable. As such, this mechanism forbids the execution of **B1** in the fifth iteration and in subsequent iterations. From the fifth to the ninth iteration, the only possible path is by following the **else** branch of **if (i == 1)** statement. This leads to the timing of 45 at the end.

At first glance, the second blocking mechanism seems to be able to make use of the provided assertion in order to block paths and tighten the bound. However, such analysis is *unsound*. A counter-example can be achieved by replacing `if (*)` with `if (prime(i))`, where `prime` is a function which returns *true* if the input is actually a *prime* number and *false* otherwise. This counter-example has the timing of $(1+5+10+10+1+10+1+10+1 = 49)$.

In summary, it is non-trivial for loop unrolling to be both *sound* and *compliant* with assertions. This requires the unrolling framework to be *fully path-sensitive* wrt. the variables used in the assertions. The greedy treatment of loops, via path merging, currently prevents us from being so. In other words, the challenge is how to address what in general is an intractable combinatorial problem.

Main Contribution

This paper proposes the first analysis framework that is *path-sensitive* and, at the same time, *faithful* to assertions. The traditional widely-used Implicit Path Enumeration Technique (IPET) [18] naturally supports assertions; it is, however, path-insensitive. To obtain precise analysis, the user needs to manually provide information regarding the loop bounds and infeasible paths. On the other hand, previous work [3] has shown that path-sensitive analysis with loop unrolling can be performed efficiently. However, as we have argued, supporting assertions in a loop unrolling framework is non-trivial.

We address the challenge by presenting an algorithm where the treatment of each loop is separated in two phases. Scalability, in both phases, is achieved using the concept of summarization with interpolant [16, 3]. We note here that, as programs usually contain more than one loop and also nested loops, our two phases are, in general, *intertwined*.

The first phase performs a symbolic execution where loops are unrolled efficiently. In order to control the explosion of possible paths, a merge of contexts is done at the end of every loop iteration. While this is an abstraction, it in general produces different contexts for different loop iterations. Thus this is the basis for being path-sensitive up to loops while capturing the non-uniform behavior of different loop iterations.

In this paper, in the first phase, a key addition to the method of [3] is to "slice" the tree by eliminating two kinds of paths:

1. those that are *infeasible* (detected from path-sensitivity), and
2. those that are *dominated*. More specifically, for each collection of paths that modify the variables used in assertions in the same way, only one path (in that collection) whose resource usage dominates the rest will be kept in the summarization.

In practice, therefore, what results from the first phase is a greatly simplified execution tree. The explored tree is then compactly represented as a transition system, which now also is a directed acyclic graph (DAG), each edge is labelled with a resource usage and how the assertion variables are modified.

In the second phase, we are no longer concerned with path-sensitivity of the original program, but instead are concerned only about assertions. More specifically, from the produced transition system, we need to disregard all paths *violating* the assertions. The problem is thus an instance of the classic Resource Constrained Shortest Path (RCSP) problem [17]. While this problem is NP-hard, we have demonstrated in [16] that, in general, the use of summarization with interpolant can be very effective. In Section 7, we further demonstrate this effectiveness with several WCET benchmark programs.

2. Related Work

The State-of-the-Art: Implicit Path Enumeration

Implicit Path Enumeration Technique (IPET) [18] is the state-of-the-art for path analysis in the domain of Worst Case Execution Time (WCET) analysis. IPET formulates the path analysis problem as an optimization problem over a set of frequency variables each associated with a basic block. More precisely, it starts with the control flow graph (CFG) of a program, where each node is a basic block. Program flow is then modeled as an assignment of values to *execution count variables*, each c_{entity} of them associated with a basic block of the program. The values reflect the total number of executions of each node for an execution of the program.

Each basic block entity with a count variable (c_{entity}) also has a timing variable (t_{entity}) giving the timing contribution of that part of the program to the total execution (for each time it is executed). Generally, t_{entity} is derived by some low-level analysis tool in which micro-architecture is modeled properly.

The possible program flows given by the structure of the program are modeled by using structural constraints over the frequency variables. Structural constraints can be automatically constructed using Kirchhoff's law. Because these constraints are quite simple, IPET has to rely on additional constraints, i.e., *user assertions*, to differentiate feasible from infeasible program flows. Some constraints are *mandatory*, like upper bounds on loops; while others will help tighten the final WCET estimate, like information on infeasible paths throughout the program. Some examples on discovering complex program flows and then feed them into IPET framework are [7, 9].

In the end, the WCET estimate is generated by maximizing, subject to flow constraints, the sum of the products of the execution counts and execution times:

$$\mathsf{WCET} = maximize(\sum_{\forall entity} c_{entity} \cdot t_{entity})$$

This optimization problem is handled using Integer Linear Programming (ILP) technique. Note that IPET does not find the worst-case execution path but just gives the worst-case count on each node. There is no information about the precise execution order.

In comparison with our work in this paper, *aside from accuracy*, we have two additional important advantages:

- IPET supports only *global assertions*, whereas we support both global and *local assertions*. IPET relies on the intuition that it is relatively easy for programmers to provide assertions in order to disregard certain paths from bound calculation, probably because they are the developers. We *partly* agree with this. There is flow information, which is hard to discover, while the programmers can be well aware of, a calculation framework should take into account such information to tighten the bounds. Nevertheless, we cannot expect the programmers to know *everything* about the program. For example, it is unreasonable to expect any programmer to state about a path which is infeasible due to a combination of guards scattered throughout the program. Such global knowledge is hard to deduce and could be as hard as the original path analysis problem. In short, it is reasonable to only assume the programmers to know *some local behavior* of a code fragment, but not all the global behaviors of the program paths. We elaborate on this when discussing Fig. 7 below.
- IPET cannot be extended to work for *non-cumulative* resource analysis such as memory high watermark analysis. The reason is that such analysis, even for a single path, depends on the order in which statements are executed; while in IPET formulation, such information is abstracted away. On the other hand, since our framework is path-based, adapting it for non-cumulative resource analysis is possible. This, however, is not the focus of our paper.

Symbolic Simulation with Loop Unrolling

In the domain of resource analysis, precision is of paramount importance. Originally, *precision* was addressed by symbolic execution with loop unrolling [19, 8, 12, 13]. A loop-unrolling approach which symbolically executes the program over all permitted inputs is clearly the most accurate. The obvious challenge is that this is generally not scalable. Thus *path-merging*, a form of abstract interpretation [4], is introduced to remedy this fact. It, in one hand, improves scalability; on the other hand, it seriously hampers the precision criterion.

The most recent related work is [3], which is a basis of the work in this paper. Its main technique to reduce both the depth and the breadth of the symbolic execution tree is by making use of *compounded summarizations*. This gives rise to the *superlinear* behavior of program with nested loops. That is, the number of states visited in the symbolic execution tree can be asymptotically smaller than the number of states in a concrete run. As a result, path-merging is still performed, but now sparsely only at the end of each loop iteration.

Parametric Bounds

Static resource analysis concerns with either *parametric* or *concrete* bounds. Parametric methods, e.g., [11, 14, 10], study the loops, recursions, and data structures, in order to come up with a closed, easy to understand worst-case formula. These methods are ideal to algorithmically explain the worst-case complexity of the resource usage.

But these methods, when being used to produce *concrete* bounds, in general, do not give precise enough answers (they concern with asymptotical precision only). Also, they are applicable to a smaller class of programs[5]. Furthermore, they usually focus on an individual loop or loop nest, rather than the whole program. Consequently, such techniques alone are mainly used for proving program termination.

We believe, however, that the advance in producing parametric bounds for complicated loops can indeed support *concrete* resource analysis, provided that an aggregation framework can make use of assertions.

[5] Most parametric methods are restricted to linear constraints and it is generally hard to adapt them to work on the level of the control flow graphs.

3. Motivating Examples

EXAMPLE 1 : Refer to the program in Fig. 1. Previously we have shown that loop unrolling produces *exact* timing analysis for this program. Here we show how IPET could exploit user assertions in order to achieve the same. Before proceeding, we mention that this example was highlighted in [18] to demonstrate the use of assertions in the IPET framework.

```
t = i = c = c1 = c2 = 0;
while (i < 10) {
   c++;
   if (i mod 3 == 0)
      { c1++; j *= j; t += 30; }
   else { c2++; j++; t += 1; }
   i++;
}
```

Figure 4: Assertions in IPET

Now see Fig. 4 where frequency variables c, c_1, and c_2 are instrumented. The structural constraint prescribed by the IPET method is $c = c_1 + c_2$. (In general, structural constraints can be easily extracted from the CFG.) The objective function to be maximized in the IPET formulation for this example is $(c_1 * 30 + c_2 * 1)$. In order to be exact, one could use the assertion $c \leq 10$ and $c_1 \leq 4$. Note that bounding c (i.e., the assertion $c \leq 10$) is *mandatory* because a bound on the objective function depends on this. On the other hand, the assertion $c_1 \leq 4$ is *optional*. Computing the optimal now will produce the exact timing.

This example also exemplifies the fact that IPET is perfectly suited to assertions simply because one just needs to conjoin the assertions to the structural constraints before performing the optimization.

It is certainly not the case that assertions alone are sufficient in general. Let us now slightly modify the example by replacing `j *= j` with `i *= i`. The new program is shown in Fig. 5.

```
t = i = c = c1 = c2 = 0;
while (i < 10) {
   c++;
   if (i mod 3 == 0)
      { c1++; i *= i; t += 30; }
   else { c2++; j++; t += 1; }
   i++;
}
```

Figure 5: Assertions Alone Are Not Enough

Following the unrolling technique, *exact* bound is still achieved. The reason is that context propagation is performed precisely. However, it is now *hard* for the user of IPET framework to come up with assertions on frequency variables in order to achieve some good bound. This scenario also poses a big challenge for many analytical methods [11, 14, 10], due to the *non-linear* operation on i, i.e., the statement `i *= i;`.

With this example, our purpose is *not to* refute the usefulness of assertions. Instead, we want to further emphasize the ability of an analyzer in propagating flow information precisely and automatically, i.e., the ability of being path-sensitive.

EXAMPLE 2 : Let us refer back to the example in Fig. 2. Now our focus is on how frequency variables and assertions should be instrumented.

First, note that our frequency variables are similar to frequency variables in IPET framework. One frequency variable is attached to a *distinct* basic block. Each is initialized to 0 at the beginning of the

```
t = c = c1 = 0;
for (i = 0; i < 100; i++) {
   c++;
   if (A[i] != 0) {
      c1++;
      t += 1000;
   } else { t += 1; }
}
assert(c1 <= c / 10);
```

Figure 6: Assertions Are Essential

program, and incremented by 1 whenever the corresponding basic block is executed. An important difference from IPET is that our frequency variables can be *reset*. This gives rise to the use of *local assertions*, and we elaborate on this below.

Our assertions are predicates over frequency variables and for simplicity, will only be provided at the end of some loop body or at the end of the program (otherwise a preprocessing phase is needed).

In Fig. 6, the assertion captures the fact that the input array A is a *sparse* one: no more 10 percent of its elements are non-zero.

```
for (i = N-1; i >= 1; i--) {
   c = 0;
   for (j = 0; j <= i-1; j++)
      if (a[j] > a[j+1]) {
         c++; t += 100; tp = a[j];
         a[j] = a[j+1]; a[j+1] = tp;
      } else { t += 1; }
      assert(c <= N-M);
   }}
```

Figure 7: Local Assertions

EXAMPLE 3 : Consider the following "bubblesort" example in Fig. 7 where we have placed a frequency variable c. An integer array `a[]` of size $N > 0$ is the input. Every element of `a[]` belongs to the integer interval $[min, max]$. Now, assume that we know that there are M elements which are equal to max. Consider: how many times are a pair of elements swapped? We believe this is currently beyond any systematic approach.

However, it is relatively easy to derive the assertion shown in the inner loop, representing *local reasoning*: each swap involves an element which is not equal to max on the right, therefore after the swap such element would not be visited again in the subsequent iterations of the inner loop. Consequently, for each invocation of the inner loop, the number of swaps is no more than the number of elements which are not equal to max.

Note that the frequency variable c is reset right before each invocation of the inner loop. If this were not done, then, to find an alternative assertion to `(c <= N-M)` may not be feasible. That is, a global assertion (in this case, to assert that the total number of increments to c) is in general *much harder* to discover than a local one.

4. Preliminaries

We model a program by a *transition system*. A transition system $\mathcal{P}$ is a triple $\langle \mathcal{L}, l_0, \longrightarrow \rangle$ where $\mathcal{L}$ is the set of program points and $l_0 \in \mathcal{L}$ is the *unique* initial program point. Let $\longrightarrow \subseteq \mathcal{L} \times \mathcal{L} \times \mathit{Ops}$, where *Ops* is the set of operations, be the transition relation that relates a state to its (possible) successors by executing the operations. This transition relation models the operations that are executed when control flows from one program point to another. We restrict all (basic) operations to be either assignments or assume operations. The set of all program variables is denoted by *Vars*. An *assignment* $\mathsf{x} := \mathsf{e}$ corresponds to assign the evaluation of the expression e to the variable x. In the *assume* operator, $\mathsf{assume(c)}$, if the conditional expression c evaluates to *true*, then the program continues, otherwise it halts. We shall use $\ell \xrightarrow{\mathsf{op}} \ell'$ to denote a transition relation from $\ell \in \mathcal{L}$ to $\ell' \in \mathcal{L}$ executing the operation $\mathsf{op} \in \mathit{Ops}$.

A transition system naturally constitutes a directed graph, where each node represents a program point and edges are defined by the relation $\longrightarrow$. This graph is similar to (but not exactly the same as) the control flow graph of a program.

One advantage of representing a program using transition systems is that the program can be executed symbolically in a simple manner. Moreover, as this representation is general enough, retargeting (e.g., to different types of applications) is just the matter of compilation to the designated transition systems.

DEFINITION 1 (Symbolic State). *A* symbolic state s *is a triple* $\langle \ell, \sigma, \Pi \rangle$*, where* $\ell \in \mathcal{L}$ *corresponds to the* concrete *current* program point*, the* symbolic store σ *is a function from program variables to terms over input symbolic variables, and the* path condition Π *is a first-order logic formula over the symbolic inputs which accumulates constraints the inputs must satisfy in order for an execution to follow the corresponding path.*

Let $s_0 \equiv \langle \ell_0, \sigma_0, \Pi_0 \rangle$ denote the unique *initial* symbolic state. At s_0 each program variable is initialized to a fresh input symbolic variable. For every state $s \equiv \langle \ell, \sigma, \Pi \rangle$, the *evaluation* $[\![e]\!]\sigma$ of an arithmetic expression e in a store σ is defined as usual: $[\![v]\!]\sigma = \sigma(v)$, $[\![n]\!]\sigma = n$, $[\![e + e']\!]\sigma = [\![e]\!]\sigma + [\![e']\!]\sigma$, $[\![e - e']\!]\sigma = [\![e]\!]\sigma - [\![e']\!]\sigma$, etc. The evaluation of conditional expression $[\![c]\!]\sigma$ can be defined analogously. The set of first-order logic formulas and symbolic states are denoted by *FO* and *SymStates*, respectively.

DEFINITION 2 (Transition Step). *Given* $\langle \mathcal{L}, \ell_0, \longrightarrow \rangle$*, a transition system, and a state* $s \equiv \langle \ell, \sigma, \Pi \rangle \in$ SymStates*, the symbolic execution of transition* $t : \ell \xrightarrow{op} \ell'$ *returns another symbolic state* s' *defined as:*

$$s' \triangleq \begin{cases} \langle \ell', \sigma, \Pi \wedge c \rangle & \text{if op} \equiv \mathsf{assume(c)} \\ \langle \ell', \sigma[x \mapsto [\![e]\!]\sigma], \Pi \rangle & \text{if op} \equiv \mathsf{x} := \mathsf{e} \end{cases} \quad (1)$$

Abusing notation, the execution step from s to s' is denoted as $s \xrightarrow{t} s'$. Given a symbolic state $s \equiv \langle \ell, \sigma, \Pi \rangle$ we also define $[\![s]\!]$: *SymStates* $\rightarrow$ *FO* as the projection of the formula $(\bigwedge_{v \in \mathit{Vars}} v = [\![v]\!]\sigma) \wedge [\![\Pi]\!]\sigma$ onto the set of program variables *Vars*. The projection is performed by the elimination of existentially quantified variables.

For convenience, when there is no ambiguity, we just refer to the symbolic state s using the pair $\langle \ell, [\![s]\!] \rangle$, where $[\![s]\!]$ is the *constraint* component of the symbolic state s. A path $\pi \equiv s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_m$ is *feasible* if $s_m \equiv \langle \ell, [\![s_m]\!] \rangle$ and $[\![s_m]\!]$ is satisfiable. Otherwise, the path is called *infeasible* and s_m is called an *infeasible* state. Here we query a *theorem prover* for satisfiability checking on the path condition. We assume the theorem prover is sound but not complete. If $\ell \in \mathcal{L}$ and there is no transition from ℓ to another program point, then ℓ is called the *ending point* of the program. Under that circumstance, if s_m is feasible then s_m is called *terminal* state.

The set of program variables *Vars* also include the resource variable r. Our formulation targets cumulative resource usage such as time, power. Thus the purpose of our path analysis is to compute a *sound* and *accurate* bound for r in the end, across all feasible paths of the program. Note that r is always initialized to 0 and the only operations allowed upon it are *concrete* increments. We assume these concrete numbers are given at the beginning of our path analysis. For instance, in WCET analysis, the amount of increment at each point will be given by some *low-level analysis* module (e.g., [23]). The resource variable r is not used in any other way.

EXAMPLE 4 : Consider the program fragment in Fig. 8(a). The program points are enclosed in angle brackets. Some of the

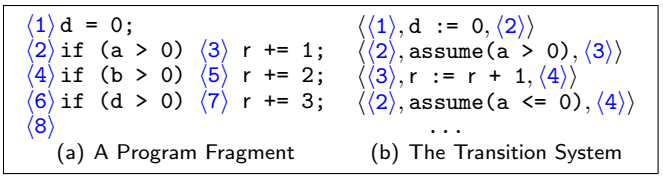

Figure 8: From a C Program to its Transition System

transitions are shown in Fig. 8(b). For instance, the transition $\langle\langle 1\rangle, \texttt{d := 0}, \langle 2\rangle\rangle$ represents that the system state switches from program point $\langle 1\rangle$ to $\langle 2\rangle$ executing the operation `d := 0`.

Recall that our transition system is a directed graph. We now introduce concepts which are required in our loop unrolling framework.

DEFINITION 3. [Loop]. *Given a directed graph $G = (V, E)$ (our transition system), we call a strongly connected component $S = (V_S, E_S)$ in G with $|E_S| > 0$, a* loop *of G.*

DEFINITION 4. [Loop Head]. *Given a directed graph $G = (V, E)$ and a loop $L = (V_L, E_L)$ of G, we call $\mathcal{E} \in V_L$ a* loop head *of L, also denoted by $\mathcal{E}(L)$, if no node in V_L, other than $\mathcal{E}$ has a direct successor outside L.*

DEFINITION 5. [Ending Point of Loop Body]. *Given a directed graph $G = (V, E)$, a loop $L = (V_L, E_L)$ of G and its loop head $\mathcal{E}$. We say that a node $u \in V_L$ is an* ending point of L's body *if there exists an edge $(u, \mathcal{E}) \in E_L$.*

We assume that each loop has only one loop head and one unique ending point. For each loop, following the back edge from the ending point to the loop head, we do not execute any operation. This can be achieved by a preprocessing phase.

DEFINITION 6. [Same Nesting Level]. *Given a directed graph $G = (V, E)$ and a loop $L = (V_L, E_L)$, we say two nodes u and v are in the* same nesting level *if for each loop $L = (V_L, E_L)$ of G, $u \in V_L \iff v \in V_L$.*

DEFINITION 7. [Assertion]. *An* assertion *is a tuple $\langle \ell, \phi(\tilde{c})\rangle$, where ℓ is a program point and $\phi(\tilde{c})$ is a set of constraints over the frequency variables $\tilde{c}$.*

DEFINITION 8. [Blocked State]. *Given a feasible state $s \equiv \langle \ell, [\![s]\!]\rangle$ and an assertion $\mathcal{A} = \langle \ell, \phi(\tilde{c})\rangle$, we say state s is* blocked *by assertion $\mathcal{A}$ if $[\![s]\!] \wedge \phi(\tilde{c})$ is unsatisfiable.*

The construction of correct summarizations requires the concept of Craig *interpolant* [5]. Examples to illustrate the following concepts will be presented in the next Section.

DEFINITION 9. [Interpolant]. *Given two first-order logic formulas F and G such that $F \models G$, then there exists a Craig* interpolant *H denoted as $Intp(F, G)$, which is a first-order logic formula such that $F \models H$ and $H \models G$, and each variable of H is a variable of both F and G.*

DEFINITION 10. [Summarization of a Subtree]. *Given two program points ℓ_1 and ℓ_2 such that ℓ_2 post-dominates ℓ_1 and assume we analyze all the paths from entry point ℓ_1 to exit point ℓ_2 wrt. an incoming context $[\![s]\!]$. The summarization of this subtree is defined as the tuple $[\ell_1, \ell_2, \Gamma, \Delta, \overline{\Psi}]$, where Γ is a set of* representative paths*, Δ is an* abstract transformer *capturing the abstract input-output relation between variables at ℓ_1 and ℓ_2, and finally, $\overline{\Psi}$ is an* interpolant*, i.e., a condition under which this summarization can be safely reused.*

Let Θ be the weakest condition such that if we examine the subtree with Θ as the incoming context, all infeasible and blocked states discovered by the previous analysis (using context $[\![s]\!]$) are preserved. As is well known, computing the weakest precondition [6] in general is expensive. The interpolant $\overline{\Psi}$ is indeed an efficiently computable approximation of Θ. Specifically, we can define $\overline{\Psi}$ as $Intp(\Theta, [\![s]\!])$.

By definition, the abstract transformer Δ [3] will be the abstraction of all feasible paths (wrt. the incoming context $[\![s]\!]$) from ℓ_1 to ℓ_2. In our implementation, we compute the abstract transformer using the polyhedral domain. Note here that, in general, abstract transformer is *not* a functional relation.

All the feasible paths of the subtree at hand are divided into a number of classes, each modifying the frequency variables in a distinct way. We are interested in only frequency variables which are *live* at the exit point ℓ_2 and will be used later in some assertion. We call those frequency variables the *relevant* ones.

Now for each class, only the *dominating* path — the one with highest resource consumption — will be kept in Γ. Specifically, each representative path $\gamma \in \Gamma$ is of the form $\langle \mathsf{r}_0, \delta_0(\tilde{c}, \tilde{c}')\rangle$, where r_0 is the amount of resource consumed in that path and $\delta_0(\tilde{c}, \tilde{c}')$ captures how frequency variables are modified in that path.

The fact that two representative paths $\gamma_1 = \langle \mathsf{r}_1, \delta_1(\tilde{c}, \tilde{c}')\rangle$ and $\gamma_2 = \langle \mathsf{r}_2, \delta_2(\tilde{c}, \tilde{c}')\rangle$ modify the set of (relevant) frequency variables in the same way is denoted by $\delta_1(\tilde{c}, \tilde{c}') \overset{\mathcal{A}}{\equiv} \delta_2(\tilde{c}, \tilde{c}')$.

DEFINITION 11. [Summarization of a Program Point]. *A summarization of a program point ℓ is the summarization of all paths from ℓ to ℓ' (wrt. the same context), where ℓ' is the nearest program point that post-dominates ℓ s.t. ℓ' is of the same nesting level as ℓ and either is (1) an ending point of the program, or (2) an ending point of some loop body.*

As ℓ' can always be deduced from ℓ, in the summarization of program ℓ, we usually omit the component about ℓ'.

5. The Algorithm: Overview of the Two Phases

Phase 1: The first phase uses a greedy strategy in the unrolling of a loop. This unrolling explores a conceptually symbolic execution tree, which is of enormous size. One main purpose of this phase is to precisely propagate the context across loop iterations, and therefore disregard as many infeasible paths as possible from consideration in the second phase.

For each iteration, for all feasible paths discovered, we divide them into a number of classes. Paths belong to a class modify the frequency variables in the same way. Paths from different classes modify the frequency variables in different ways. As an optimization, we are only interested in frequency variables which will later be used in some assertion. For each class, only the path with highest resource consumption is kept, we say that path is the *dominating* path of the corresponding class. Thus another purpose of the first phase is to disregard dominated paths from consideration in the second phase.

From the dominating paths discovered, we now represent compactly this iteration as a set of transitions. This representation is manageable because we can restrict attention only to the frequency variables used later in some assertion. We continue this process iteration by iteration. For two different iterations, the infeasible paths detected in each iteration can be quite different. As a result, their representations will be different too. At the end of phase 1, we represent the unrolled loop in the form of a transition system in order to avoid an upfront consideration of the search space for the whole loop, which can potentially still be exponential.

EXAMPLE 5 : Consider the program fragment in Fig. 9, which is slightly modified from the example shown in Fig. 3 (and now with instrumented program points). Note that at phase 1, we ignore the assertion at program point $\langle 11\rangle$, but pay attention only to its frequency variable c.

```
⟨1⟩   c = 0, i = 0, t = 0;
⟨2⟩   while (i < 9) {
⟨3⟩      if (*) {
⟨4⟩         if (*) {
⟨5⟩            c++; t += 10;
            } else {
⟨6⟩            t += 1;
            }
         } else {
⟨7⟩         if (i == 1) {
⟨8⟩            t += 5;
            } else {
⟨9⟩            t += 1;
            }
         }
⟨10⟩     i++;
⟨11⟩     assert(c <= 4);
⟨12⟩  }
⟨13⟩
```

Figure 9: Complying with Assertions in Loop Unrolling

We enter the first iteration of the loop. Inside the loop body, we follow the first feasible path (⟨3⟩ ⟨4⟩ ⟨5⟩ ⟨10⟩ ⟨12⟩). The value of i at ⟨12⟩ is 1. When backtracking, a summarization of program point ⟨10⟩ is computed as:

$$[\langle 10\rangle, \{\langle 0, \mathsf{c} := \mathsf{c}\rangle\}, i' = i + 1, \mathit{true}]$$

In other words, the subtree from ⟨10⟩ to ⟨12⟩ is summarized by: (1) a representative path which does not consume any resource and does not modify the assertion variable c either; (2) an abstract transformer which says that the output value of i is equal to the input value of i plus 1, (3) an interpolant true which means that any state at program point ⟨10⟩ can safely reuse this summarization. Similarly, we derive a summarization for program point ⟨5⟩ as:

$$[\langle 5\rangle, \{\langle 10, \mathsf{c} := \mathsf{c}{+}1\rangle\}, i' = i + 1, \mathit{true}]$$

The main difference here is that the representative path from ⟨5⟩ to ⟨12⟩ consumes 10 units of time and increments the assertion variable c by 1. From ⟨4⟩, we now follow the **else** branch of the second **if-statement** to reach ⟨6⟩ and then ⟨10⟩. At ⟨10⟩ we reuse the computed summarization for ⟨10⟩. The abstract transformer $i' = i + 1$ is used to produce the continuation context for i at ⟨12⟩ ($i = 1$). We then backtrack and a summarization for ⟨6⟩ is computed as:

$$[\langle 6\rangle, \{\langle 1, \mathsf{c} := \mathsf{c}{+}1\rangle\}, i' = i + 1, \mathit{true}]$$

Thus the combined summarization for ⟨4⟩ (from ⟨5⟩, ⟨6⟩) is:

$$[\langle 4\rangle, \{\langle 10, \mathsf{c} := \mathsf{c}{+}1\rangle, \langle 1, \mathsf{c} := \mathsf{c}\rangle\}, i' = i + 1, \mathit{true}]$$

Note that this summarization of ⟨4⟩ contains two representative paths, since there are two distinct ways in modifying the assertion variable c. From ⟨3⟩ we now follow the **else** branch of the first **if-statement**. Since i is currently 0, going from ⟨7⟩ to ⟨8⟩ (the **then** branch of the third **if-statement**) is infeasible. This fact is summarized as:

$$[\langle 8\rangle, \emptyset, \mathit{false}, \mathit{false}]$$

Following the **else** branch we reach ⟨9⟩ and then ⟨10⟩. At ⟨10⟩ we reuse and backtrack. The summarization of ⟨9⟩ is:

$$[\langle 9\rangle, \{\langle 1, \mathsf{c} := \mathsf{c}\rangle\}, i' = i + 1, \mathit{true}]$$

Thus the combined summarization for ⟨7⟩ (from ⟨8⟩, ⟨9⟩) is:

$$[\langle 7\rangle, \{\langle 1, \mathsf{c} := \mathsf{c}\rangle\}, i' = i + 1, i \neq 1]$$

Note how the infeasible paths from ⟨7⟩ to ⟨8⟩ affects the interpolant for the summarization at ⟨7⟩. Now we need to combine the summarizations of ⟨4⟩ and ⟨7⟩ to get a summarization for ⟨3⟩. We can see that the second representative path in the summarization of ⟨4⟩ and the only representative path in the summarization of ⟨7⟩ both do not modify the frequency variable c and consume 1 unit of time. In other words, each of them dominates the other. Consequently, we only keep one of them in the summarization of ⟨3⟩. The interpolants for ⟨4⟩ and ⟨7⟩ are propagated back (we use precondition computation) and conjoined to give the interpolant for ⟨3⟩. The summarization of ⟨3⟩ wrt. the context of the first iteration of the loop is then:

$$[\langle 3\rangle, \{\langle 10, \mathsf{c} := \mathsf{c}{+}1\rangle, \langle 1, \mathsf{c} := \mathsf{c}\rangle\}, i' = i + 1, i \neq 1]$$

For the first iteration, we add into our new transition system (we omit $\mathsf{c} := \mathsf{c}$ in the second transition):

$$\langle\langle 2\rangle\text{-}0\rangle, \mathsf{c} := \mathsf{c}{+}1 \wedge \mathsf{t} := \mathsf{t}{+}10, \langle\langle 2\rangle\text{-}1\rangle\rangle$$
$$\langle\langle 2\rangle\text{-}0\rangle, \mathsf{t} := \mathsf{t}{+}1, \langle\langle 2\rangle\text{-}1\rangle\rangle$$

The second iteration begins with the context $i = 1$. At program point ⟨3⟩, as the current context does not imply the interpolant $i \neq 1$ of the existing summarization for ⟨3⟩, reuse does not happen. Follow the **then** branch, we reach program point ⟨4⟩ and we can reuse the existing summarization of ⟨4⟩, also produce a continuation context $i = 2$ using the abstract transformer $i' = i + 1$. We then visit program point ⟨7⟩ where we cannot reuse previous analysis. Different from the first iteration, going from ⟨7⟩ to ⟨8⟩ is now feasible while going from ⟨7⟩ to ⟨9⟩ is infeasible. As a result, a new summarization for ⟨7⟩ is computed as:

$$[\langle 7\rangle, \{\langle 5, \mathsf{c} := \mathsf{c}\}\rangle, i' = i + 1, i = 1]$$

Subsequently, a summarization of ⟨3⟩ wrt. the context of the second iteration is computed as:

$$[\langle 3\rangle, \{\langle 10, \mathsf{c} := \mathsf{c}{+}1\rangle, \langle 5, \mathsf{c} := \mathsf{c}\}\rangle, i' = i + 1, i = 1]$$

For the second iteration, we add in the following transitions:

$$\langle\langle 2\rangle\text{-}1\rangle, \mathsf{c} := \mathsf{c}{+}1 \wedge \mathsf{t} := \mathsf{t}{+}10, \langle\langle 2\rangle\text{-}2\rangle\rangle$$
$$\langle\langle 2\rangle\text{-}1\rangle, \mathsf{t} := \mathsf{t}{+}5, \langle\langle 2\rangle\text{-}2\rangle\rangle$$

Analyses of subsequent iterations reuse the analysis of the first iteration (since the contexts imply the interpolant $i \neq 1$). The following transitions — from iteration j to iteration $j + 1$, where $j = 2..8$ — will be added into the new transition system. Note that we also add the last transition which corresponds to the loop exit.

$$\langle\langle 2\rangle\text{-}2\rangle, \mathsf{c} := \mathsf{c}{+}1 \wedge \mathsf{t} := \mathsf{t}{+}10, \langle\langle 2\rangle\text{-}3\rangle\rangle$$
$$\langle\langle 2\rangle\text{-}2\rangle, \mathsf{t} := \mathsf{t}{+}1, \langle\langle 2\rangle\text{-}3\rangle\rangle$$
$$\cdots$$
$$\langle\langle 2\rangle\text{-}8\rangle, \mathsf{c} := \mathsf{c}{+}1 \wedge \mathsf{t} := \mathsf{t}{+}10, \langle\langle 2\rangle\text{-}9\rangle\rangle$$
$$\langle\langle 2\rangle\text{-}8\rangle, \mathsf{t} := \mathsf{t}{+}1, \langle\langle 2\rangle\text{-}9\rangle\rangle$$
$$\langle\langle 2\rangle\text{-}9\rangle, , \langle 13\rangle\rangle$$

We note here that the representation of the second iteration is different from the representations of all other iterations. This arises from the fact that the iterations of the loop do not behave uniformly. This can only be captured by loop unrolling while being path-sensitive.

Before proceeding, we first comment that the simplification that phase 1 performs is often very significant. This is essentially because each iteration can exploit the path-sensitivity that survives through the merges of previous iterations. In general, this leads to an exponential decline in the total number of paths in the resulting transition system. Importantly, now phase 2 (in which no path-merging is performed) only needs to track the assertion variables in order to disregard paths violating the provided assertions from consideration. Without phase 1, phase 2 would have to consider also all program variables. This would make the usage of summarization with interpolant much less effective.

Return to the above example, for each iteration, we have reduced the number of paths to be considered in the second phase from 4 to 2. Also, phase 2 now only needs to track the assertion variable c in order to block invalid paths from consideration.

Phase 2: In the second phase, we attack the remaining problem, to determine the longest path in this new transition system, also using the concept of summarization with interpolant. The key point to note is that only at this second phase, assertions are taken into consideration to block paths. Consequently, paths violating the assertions will be considered as infeasible, i.e., they are disregarded from bound calculation.

Let us continue with Ex. 5. Consider the transition system from phase 1. We now need to solve the longest path problem using the initial context $(c = 0, t = 0)$. The original assertion will be checked at every program point $\langle\langle 2\rangle\text{-j}\rangle$ for $j = 1..9$. To be faithful to the given assertion, we must disregard all paths which increment c more than 4 times from consideration.

In phase 2, our analysis using summarization with interpolant is now performed on a new transition system which contains no loops. Given the transition system produced by phase 1, a naive method would require to explore 894 states from 2^9 paths. By employing summarizations with interpolants the number of explored states is reduced to 56, of which 24 states are reuse states. The effectiveness of summarization with interpolant for such problem instances has already been demonstrated in [16].

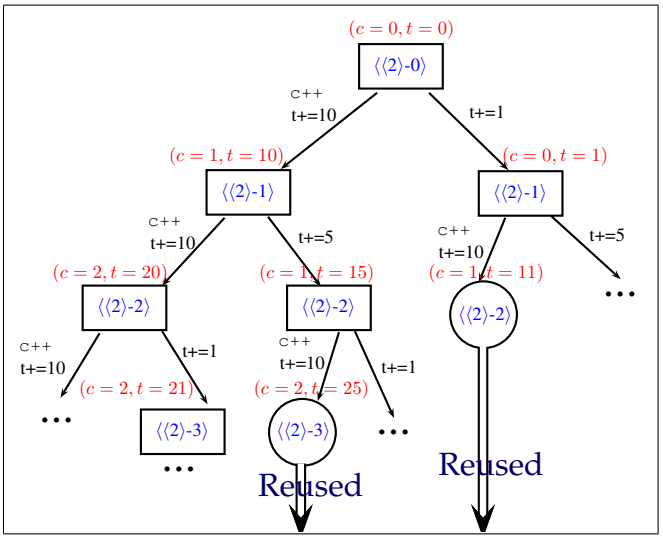

Figure 10: The Search Tree Explored in Phase 2

Instead of walking through (again) the application of summarization with interpolant, we offer some insights as to why phase 2 is often tractable (though the expanded tree is still quite deep). See Fig. 10, where circles denote "reuse" states and require no further expansion. Note that although the DAG contains exponentially many paths, there are only a few contexts of interest, namely $c = \alpha$ where $0 \leq \alpha \leq 9$. Therefore, for each node in the DAG there needs only 10 considerations of paths starting from the node. Thus a straightforward dynamic programming approach would suffice. However, it is important to note that in general (for example, more than one frequency variable), the number of different contexts of each node is exponential. The algorithm we use has the special advantage of using interpolation so that the dynamic programming effect can be enjoyed by considering not just the context, but some sophisticated *generalization* of the context. Essentially, two contexts can in fact be considered equal if they exhibit the same infeasible/blocked paths.

6. The Algorithm: Technical Description

We proceed to phase 2 with this new transition system, $\mathcal{G}$, as in line 4. The worst case bound is then achieved by looking for the maximum value in all returned solution paths Γ (line 5).

```
function Analyze(s_0, P)
⟨1⟩  [·, Γ_0, ·, ·] := Summarize(s_0, P, 1)
⟨2⟩  G := ∅
⟨3⟩  G := Build(Γ_0, P, G)
⟨4⟩  [·, Γ, ·, ·] := Summarize(s_0, G, 2)
⟨5⟩  return FindMax(Γ)

function Summarize(s, P, phase)
     Let s be ⟨ℓ, [[s]]⟩
⟨6⟩  if ([[s]] ≡ false) return [ℓ, ∅, false, false]
⟨7⟩  if (outgoing(ℓ, P) ≡ ∅) return [ℓ, {⟨0, Id(c̃)⟩}, Id(Vars), true]
⟨8⟩  if (loop_end(ℓ, P)) return [ℓ, {⟨0, Id(c̃)⟩}, Id(Vars), true]
⟨9⟩  S := memoed(s)
⟨10⟩ if (S ≢ false) return S
⟨11⟩ if (phase ≡ 2)                    /* Consider assertions at phase 2 */
⟨12⟩   if (∃ A ≡ ⟨ℓ, φ⟩ and [[s]] ∧ φ ≡ false) return [ℓ, ∅, false, ¬φ]
⟨13⟩ if (phase ≡ 1 ∧ loop_head(ℓ, P))
⟨14⟩   s_i := s
⟨15⟩   G := ∅
⟨16⟩   [·, Γ_1, Δ_1, ·] := TransStep(s_i, P, entry(ℓ, P), 1)
⟨17⟩   while (Γ_1 ≢ ∅)
⟨18⟩     G := Build(Γ_1, P, G)
⟨19⟩     [·, Γ_2, ·, ·] := TransStep(s_i, P, exit(ℓ, P), 1)
⟨20⟩     if (Γ_2 ≢ ∅) G := Build(Γ_2, P, G)
⟨21⟩     s_i --Δ_1--> s'_i             /* Execute abstract transition Δ_1 */
⟨22⟩     s_i := s'_i
⟨23⟩     [·, Γ_1, Δ_1, ·] := TransStep(s_i, P, entry(ℓ, P), 1)
         endwhile
⟨24⟩   [·, Γ_2, ·, ·] := TransStep(s_i, P, exit(ℓ, P), 1)
⟨25⟩   G := Build(Γ_2, P, G)
⟨26⟩   S := Summarize(s, G, 2)                          /* Phase 2 */
⟨27⟩ else S := TransStep(s, P, outgoing(ℓ, P), phase)
⟨28⟩ if (phase ≡ 2) S̄ := Modification of S
           taking into account the information computed in phase 1
⟨29⟩ else S̄ := S
⟨30⟩ memo and return S̄
```

Figure 11: Two-phase Symbolic Simulation Algorithm

Our *key* function, Summarize, takes as inputs a symbolic state $s \equiv \langle \ell, [\![s]\!] \rangle$, a transition system, and a flag indicating which phase it is in. It then performs the analysis using the context $[\![s]\!]$ and returns the summarization for the program point ℓ as in Def. 11 in Section 4.

Base Cases: Summarize handles 4 base cases. First, when the symbolic state s is infeasible (line 6), no execution needs to be considered. Note that here path-sensitivity plays a role since only provably executable paths will be considered. Second, s is a terminal state (line 7). Here Id refers to the identity function, which keep the values of variables unchanged. Ending point of a loop is treated similarly in the third base case (line 8). The last base case, lines 9-10, is the case that a summarization can be reused. We have demonstrated this step, with examples, in Section 5.

Expanding to next Program Points: Line 27 depicts the case when transitions can be taken from current program point ℓ, and ℓ is not a loop starting point. Here we call TransStep to move recursively to next program points. TransStep implements the traversal of transition steps emanating from ℓ, denoted by outgoing$(\ell, \mathcal{P})$, by calling Summarize recursively and then compounds the returned summarizations into a summarization of ℓ. The inputs of TransStep are symbolic state s, the transition system $\mathcal{P}$, a set of outgoing transitions *TransSet* to be explored, and the current phase the algorithm is in.

For each t in *TransSet*, TransStep extends the current state with the transition. Resulting child state is then given as an argument in a recursive call to Summarize (line 34). From each summarization of a child returned by the call to Summarize, the algorithm computes a component summarization, contributed by that particular child to the parent as in lines 35-39. All of such components will be compounded using the JoinHorizontal function (line 40).

Note that the interpolant for the child state is propagated back to its parent using the *precondition operation* pre, where $\mathsf{pre}(t, \overline{\Psi})$ denotes the precondition of the postcondition $\overline{\Psi}$ wrt. the transition t. In an ideal case, we would want this operation to return the *weakest precondition*. But in general that could be too expensive. Discussions on possible implementations of this operator can be found at [21, 3]. In our implementation using CLP($\mathcal{R}$) [15], the combine function simply conjoins the corresponding constraints and performs projections to reduce the size of the formula.

Loop Handling: Lines 13-26 handle the case when the current program point ℓ is a loop head. Let $\mathsf{entry}(\ell, \mathcal{P})$ denote the set of transitions going into the body of the loop, and $\mathsf{exit}(\ell, \mathcal{P})$ denote the set of transitions exiting the loop.

Upon encountering a loop, our algorithm attempts to unroll it once by calling the function TransStep to explore the entry transitions (line 16). When the returned set of representative paths is empty, it means that we cannot go into the loop body anymore, we thus proceed to the exit transitions (lines 24-25). Otherwise, if some feasible paths are found by going into the loop body, we first use the returned set of representative paths Γ_1 to add new transitions into our transition system $\mathcal{G}$ (line 18). Next we use the returned abstract transformer to produce a new continuation context (lines 21-22), so that we can continue the analysis with the next iteration. Here we assume that this unrolling process will eventually terminate. However, since we are performing symbolic execution, it is possible that at some iterations both the entry transitions and exit transitions are feasible. Lines 19-20 accommodate this fact.

Phase 2: In phase 2, we now make use of assertions, to block paths. This is achieved at lines 11-12. Note that the negation of the assertion will be the interpolant for the current state and this interpolant will be propagated backward.

The summarization of a program point ℓ at phase 2 will be modified, as in line 28. The abstract transformer of this summarization is the one computed in phase 1. However, the interpolant is combined by conjoining the interpolant of that program point already computed in phase 1 to the current interpolant in phase 2. This is because an interpolant of a node comes from two sources. The first is due the *infeasible paths* detected in phase 1, while the second is due to the *blocked paths* detected in phase 2. We note here that, for simplicity, we purposely omit the details in phase 1 on how summarizations are vertically combined, so as to produce a serialization of summarizations for the loop head. For details, see [3].

Combining Summarizations: Function Merge_Paths simply merges two sets of paths into one. As mentioned before, for each distinct way of changing the frequency variables which are relevant to some assertions used later, we only keep the dominating path and ignore all the dominated paths.

Given two subtrees T_1 and T_2 which are siblings and the inputs S_1 and S_2 summarize T_1 and T_2 respectively. JoinHorizontal is

```
      function TransStep(s, P, TransSet, phase)
            Let s be ⟨ℓ, [[s]]⟩
⟨31⟩        S̄ := [ℓ, ∅, false, true]
⟨32⟩        foreach (t ∈ TransSet ∧ t contains r := r+α) do
⟨33⟩           s --t--> s'
⟨34⟩           [ℓ', Γ, Δ, Ψ̄] := Summarize(s', P, phase)
⟨35⟩           Γ' := ∅
⟨36⟩           foreach (⟨r1, δ1⟩ ∈ Γ) do
⟨37⟩              one := {⟨r1 + α, combine(t, δ1)⟩}
⟨38⟩              Γ' := Merge_Paths(Γ', one)
               endfor
⟨39⟩           S := [ℓ, Γ', combine(t, Δ), pre(t, Ψ̄)]
⟨40⟩           S̄ := JoinHorizontal(S̄, S)
            endfor
⟨41⟩        return S̄
```

```
      function Merge_Paths(Γ1, Γ2)
⟨42⟩     Γ := Γ1
⟨43⟩     foreach (γ2 := ⟨r2, δ2(c̃, c̃')⟩ ∈ Γ2) do
⟨44⟩        status := true
⟨45⟩        foreach ( γ1 := ⟨r1, δ1(c̃, c̃')⟩ ∈ Γ) do
⟨46⟩           if ( δ1(c̃, c̃') ≡^A δ2(c̃, c̃'))
⟨47⟩              status := false
⟨48⟩              if (r2 > r1) replace γ1 in Γ by γ2
⟨49⟩              break                    /* Out of the inner loop */
           endfor
⟨50⟩        if (status) add γ2 into Γ
         endfor
⟨51⟩     return Γ
      function JoinHorizontal(S1, S2)
            Let S1 be [ℓ, Γ1, Δ1, Ψ̄1]
            Let S2 be [ℓ, Γ2, Δ2, Ψ̄2]
⟨52⟩     Γ := Merge_Paths(Γ1, Γ2)
⟨53⟩     Δ := Δ1 ∨ Δ2                  /* Merge two abstract transformers */
⟨54⟩     Ψ̄ := Ψ̄1 ∧ Ψ̄2                  /* Conjoin two interpolants */
⟨55⟩     return [ℓ, Γ, Δ, Ψ̄]
```

then used to produce the summarization of the compounded subtree T of both T_1 and T_2. Here the representative paths are merged (line 52). Preserving all infeasible/blocked paths in T requires preserving infeasible/blocked paths in both T_1 and T_2 (line 54). The input-output relationship of T is safely abstracted as the disjunction of the input-output relationships of T_1 and T_2 respectively (line 53). In our implementation, this corresponds to the convex hull operator of the polyhedral domain.

7. Experimental Evaluation

We used a 2.93Gz Intel processor and 2GB RAM. Our prototype is implemented in C, while making use of the arithmetic solver of CLP($\mathcal{R}$)[15]. Our polyhedral library is the same as in [3], interested readers might refer to [3] for more details.

Benchmark	LOC	Path-Sensitive				Path-Insensitive (IPET)	
		w.o. Assertions		w. Assertions		w.o. As	w. As
		Bound	T(s)	Bound	T(s)		
Ex-2	<100	110404	1.50	10404	3.48	110404	10404
Ex-3	<100	515398	5.52	**49798**	11.45	1019902	1019902
Ex-5	<100	1504	3.47	**759**	9.22	1504	1129
insertsort100	<100	515794	4.91	**30802**	7.78	1020804	1020804
crc	128	1404	7.73	1084	8.61	1404	1084
expint	157	15709	4.40	**859**	4.56	-	-
matmult100	163	3080505	4.55	131705	5.54	3080505	131705
fir	276	1129	2.35	**793**	2.39	-	-
fft64	219	7933	5.52	**1733**	6.04	-	-
tcas	400	159	3.84	**81**	3.9	172	94
statemate	1276	2103	9.65	**1103**	9.73	2271	1271
nsichneu_s	2334	483	9.43	**383**	9.51	2559	2459

Table 1. Experiments with and without Assertions

We evaluate our framework on the problem of WCET analysis. The programs used for evaluation are: (1) academic examples presented in this paper; (2) benchmarks from Mälardalen WCET group [20], which are often used for evaluations of WCET path analysis techniques; and (3) a real traffic collision avoidance system `tcas`. Their corresponding sizes (LOC) are given in the second column of Table 1.

We have thoroughly argued the benefits of being path-sensitive while compliant with user assertions. The aim of this Section is to demonstrate the scalability of our two-phase algorithm. Table 1 shows the experimental results. For each benchmark, the worst case (low-level) timing for each basic block is randomly generated. Assertions are simply to bound the frequencies of some basic blocks having (relatively) substantial worst case timings (so that the effects

can be shown). Being path-sensitive does not help automatically infer such bounds, i.e., such manually given assertions are necessary.

We evaluate the effects of assertions in our path-sensitive framework as well as in IPET, a path-insensitive framework. We remark that IPET is the current state-of-the-art in WCET path analysis and is used in most available WCET tools (e.g., aiT [1]).

The last two columns are about IPET. Recall that IPET *always* requires assertions on loop bounds in order to produce an answer. For programs where such information can be easily extracted from the code, we provide IPET such loop bounds. Loop bounds which must be dynamically computed (e.g., from unrolling) are not provided to IPET. As a result, IPET cannot produce bounds for some programs, indicated as '-'. For programs where IPET can successfully bound, IPET running time is always less than 1 second, so we do not tabulate those timings individually.

The third and fourth columns show the bounds and the running time using our unrolling framework without employing assertions. These results correspond to the results produced by [3], which is representative for the state-of-the-arts in loop unrolling. On the other hand, the fifth and sixth columns report the performance of the algorithm proposed in this paper, possessing path-sensitivity as well as being compliant with assertions. As expected, our algorithm produces the best bounds for all instances. Importantly, for most programs, it achieves more precise bounds which neither path-sensitivity alone nor user assertions alone can achieve.

Our algorithm scales well, even for programs with (nested) loops and of practical sizes. This is due to the use of compounded summarizations. From a close investigation, we see that our algorithm preserves the *superlinear* behavior observed in [3]. Also, the cost of complying with assertions, i.e., the cost of phase 2, depends mainly on the assertions and the maximum number of iterations for the loops where the assertions are used. Such cost does not depend on the size of the input program, nor the size of the overall symbolic execution tree.

In summary, we have repeated on earlier experiments [3] to demonstrate the superiority of having path-sensitivity, not considering assertions. Then we considered assertions, and demonstrated two things. Foremost is that our two-phase algorithm, which is new, can scale to practical sized programs. We also demonstrated along the way that assertions can influence the resource analysis in a significant way.

8. Concluding Remarks

We considered the problem of symbolic simulation of programs, where loops are unrolled, in the pursuit of resource analysis. A main requirement is that assertions may be used to limit possible execution traces. The first phase of the algorithm performed symbolic simulation without consideration of assertions. From this, a skeletal transition system — which now only concerns the assertions, and is much simpler than the system corresponding to the original program — is produced. The second phase now determines the worst-case path in this simplified transition system. While this problem is an instance of an NP-hard problem (RCSP), we argue that the instances arising from assertions in program analysis lends itself to efficient solution using a dynamic programming plus interpolation approach. Finally, benchmarks clearly show that the algorithm can scale.

References

[1] aiT Worst-Case Execution Time Analyzers. URL `http://www.absint.com/ait/index.htm`.

[2] Bound-T time and stack analyser. URL `http://www.bound-t.com`.

[3] D. H. Chu and J. Jaffar. Symbolic simulation on complicated loops for wcet path analysis. In *EMSOFT*, pages 319–328, 2011.

[4] P. Cousot and R. Cousot. Abstract Interpretation: A unified lattice model for static analysis. In *POPL*, pages 238–252, 1977.

[5] W. Craig. Three uses of Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.

[6] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[7] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *RTSS*, pages 163–174, 2000.

[8] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Euro-Par*, pages 1298–1307, 1997.

[9] A. Ermedahl, F. Stappert, and J. Engblom. Clustered calculation of worst-case execution times. In *CASES*, pages 51–62, 2003.

[10] D. Esteban and S. Genaim. On the limits of the classical approach to cost analysis. In *SAS*, pages 405–421, 2012.

[11] S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI*, pages 292–304, 2010.

[12] J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system C programs. In *WORDS*, pages 287–300, 2005.

[13] J. Gustafsson, A. Ermedahl, and B. Lisper. Algorithms for infeasible path calculation. In *WCET*, 2006.

[14] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. In *POPL*, pages 357–370, 2011.

[15] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS*, 14(3):339–395, 1992.

[16] J. Jaffar, A. E. Santosa, and R. Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *AAAI*, pages 297–303, 2008.

[17] H. C. Joksch. The shortest route problem with constraints. *Journal of Mathematical Analysis and Applications*, 14(2):191–197, 1966.

[18] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, pages 456–461, 1995.

[19] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2):183–207, 1999.

[20] Mälardalen WCET research group benchmarks. URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, 2006.

[21] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. In *VMCAI*, pages 346–362, 2007.

[22] A. Thakur and R. Govindarajan. Comprehensive path-sensitive data-flow analysis. In *CGO*, pages 55–63, 2008.

[23] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by seperate cache and path analyses. *Real-Time Systems*, 18(2):157–179, 2000.