Quantitative Analysis of Software: Challenges and Recent Advances

Sanjit A. Seshia

EECS Department, UC Berkeley
sseshia@eecs.berkeley.edu

Abstract. Even with impressive advances in formal methods over the last few decades, some problems in automatic verification remain challenging. Central amongst these is the verification of quantitative properties of software such as execution time or energy usage. This paper discusses the main challenges for quantitative analysis of software in cyber-physical systems. It also presents a new approach to this problem based on the combination of inductive inference with deductive reasoning. The approach has been implemented for timing analysis in a system called GAMETIME.

1 Introduction

Cyber-physical systems tightly integrate computation with the physical world. Consequently, the behavior of software controllers of such systems has a major effect on physical properties of such systems. These properties are *quantitative*, encoding specifications on physical quantities such as time, energy, position, and acceleration. The verification of such quantitative properties of cyber-physical software systems requires modeling not only the software program but also the relevant aspects of the program's environment. In contrast with traditional "Boolean" verification of software, environment models must be more precise — for example, one cannot liberally employ non-determinism in modeling the environment, and one cannot abstract away the hardware or the network. This challenge of accurate modeling is one of the major reasons why the progress on quantitative software verification has lagged behind that on Boolean software verification.

Consider, for example, the area of timing analysis of software. Several kinds of timing analysis problems arise in practice. First, for hard real-time systems, a classic problem is to estimate the worst-case execution time (WCET) of a terminating software task. Such an estimate is relevant for verifying if deadlines or timing constraints are met as well as for use in scheduling strategies. Second, for soft real-time systems, it can be useful to estimate the distribution of execution times exhibitable by a task. Third, it can be very useful to find a test case on which the program exhibits anomalous timing behavior; e.g., a test case causing a task to miss its deadline. Finally, in "software-in-the-loop" simulation, the software implementation of a controller is simulated along with a model of the continuous plant it controls, with the simulations connected using execution time estimates. For scalability, such simulation must be performed on a workstation, not on the target embedded platform. Consequently, during the workstation-based simulation, it is necessary to predict the timing of the program along a particular execution path on the target platform. All of these problems are instances of *predicting* a particular execution time property of a terminating software task.

In particular, the problem of WCET estimation has been the subject of significant research efforts over the last 20 years (e.g. [3, 4]). Significant progress has been made on this prob-

lem, especially in the computation of bounds on loops in tasks, in modeling the dependencies amongst program fragments using (linear) constraints, and modeling some aspects of processor behavior. However, as pointed out in recent papers (e.g., Lee [2]), it is becoming increasingly difficult to precisely model the complexities of the underlying hardware platform (e.g., out-of-order processors with deep pipelines, branch prediction, multi-level caches, parallelism) as well as the software environment. This results in timing estimates that are either too pessimistic (due to conservative platform modeling) or too optimistic (due to unmodeled features of the platform). Due to the difficulty of platform modeling, industry practice typically involves making random, unguided measurements to obtain timing estimates, but these provide no guarantees.

In Section 2, we elaborate on the challenge of *environment modeling*. Techniques for automatically inferring an adequate environment model are required to address this challenge. In Section 3, we present the first steps towards such solution, implemented in the timing analysis tool GAMETIME. We conclude in Section 4 with directions for future research.

2 Challenge: Environment Modeling

We discuss the challenge of environment modeling using timing analysis as an example. The complexity of the timing analysis arises from two dimensions of the problem: the *path dimension*, where one must find the worst-case computation path for the task, and the *state dimension*, where one must find the right (starting) environment state to run the task from. Moreover, these two dimensions interact closely; for example, the choice of path can affect the impact of the starting environment state.

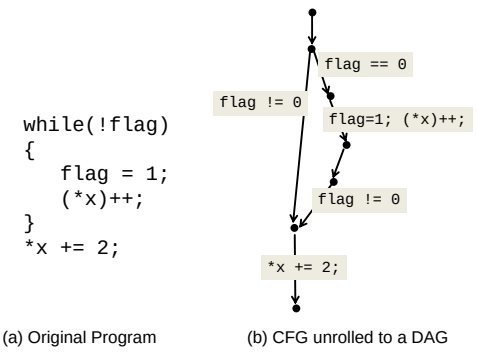

Fig. 1. Simple Illustrative Example

Consider the toy C program in Fig. 1(a). It contains a loop, which executes at most once. Thus, the control-flow graph (CFG) of the program can be unrolled into a directed acyclic graph (DAG), as shown in Fig. 1(b). Suppose we execute this program on a simple processor with an in-order pipeline and a data cache. Consider executing this program from the state where the cache is empty. The final statement of the program, `*x += 2`, contains a load, a store, and an arithmetic operation. If the left-hand path is taken, the load will suffer a cache miss; however, if the right-hand path is taken, there is a cache hit. The difference in timing between a cache hit and a miss can be an order of magnitude. Thus, the time taken by this statement depends on the program path taken. However, if the program were executed from a state with the data in the cache, there will be a cache hit even if the left-hand path is taken.

Thus, even with this toy program and a simple processor, one can observe that a timing analysis tool must explore the space of all possible program paths – a potentially exponentially large search space. If the starting environment state is known, a compositional timing tool that seeks to predict path timing by measuring timing of basic blocks must (i)

model the platform precisely to predict timing at basic block boundaries, and (ii) search an exponentially-large environment state space at these boundaries. If the starting environment state is unknown, the problem is even harder.

Current state-of-the-art tools for timing analysis rely heavily on manually-constructed abstract timing models to achieve the right balance of precision and scalability. Manual modeling can be extremely tedious and error-prone, requiring several man-months of effort to design a timing model correctly even for a simple microcontroller. The challenge of keeping up with today's advances in microarchitecture are really daunting.

3 Automatic Model Inference: The GAMETIME Approach

Automatic inductive inference of models offers a way to mitigate the challenge of environment modeling. In this approach, a *program-specific timing model* of the platform is inferred from carefully chosen observations of the program's timing. The program-specificity is an important difference from traditional approaches, which seek to manually construct a timing model that works for *all* programs one might run on the platform. The latter is a very hard problem and perhaps not one we need to necessarily solve! An accurate model for the programs of interest should suffice.

This approach has been implemented for timing analysis in a toolkit called GAMETIME [7, 6, 5]. In GAMETIME, the platform is viewed as an adversary that controls the choice and evolution of the environment state, while the tool has control of the program path space. The analysis problem is then a game between the tool and the platform. In contrast with most existing tools for timing analysis (see, e.g., [4]), GAMETIME can predict not only extreme-case behavior, but also certain execution time statistics (e.g., the distribution) as well as a program's timing along particular execution paths. Additionally, it only requires one to run end-to-end measurements on the target platform, making it easy to port to new platforms. The GAMETIME approach, along with an exposition of theoretical and experimental results, including comparisons with other methods, is described in existing papers [7, 6, 5]. We provide only a brief overview of the approach taken by GAMETIME here.

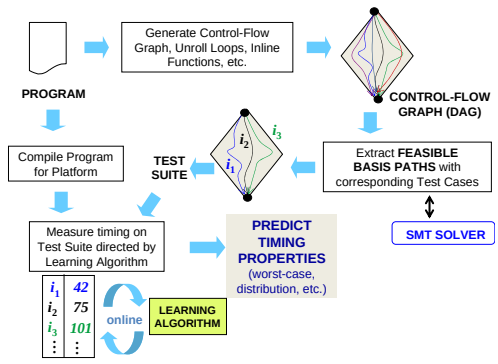

Fig. 2. GAMETIME **overview**

Figure 2 depicts the operation of GAMETIME. As shown in the top-left corner, the process begins with the generation of the control-flow graph (CFG) corresponding to the program, where all loops have been unrolled to a maximum iteration bound, and all function calls have been inlined into the top-level function. The CFG is assumed to have a single source node (entry point) and a single sink node (exit point); if not, dummy source and sink nodes are added. The next step is a critical one, where a subset of program paths, called *basis paths* are extracted. These basis paths are those that form a basis for the set of all paths, in the standard linear algebra sense of a basis. A *satisfiability modulo theories (SMT) solver*

— a deductive engine — is invoked to ensure that the generated basis paths are feasible. For each feasible basis path generated, the SMT solver generates a test case that drives program execution down that path. Thus a set of *feasible basis paths* is generated that spans the entire space of feasible program paths, along with the corresponding test cases.

The program is then compiled for the target platform, and executed on these test cases. In the basic GAMETIME algorithm (described in [7, 6]), the sequence of tests is randomized, with basis paths being chosen uniformly at random to be executed. The overall execution time of the program is recorded for each test case. From these end-to-end execution time measurements, GAMETIME's learning algorithm generates a weighted graph model that is used to make predictions about timing properties of interest. The predictions hold with high probability under certain assumptions; see the previous papers on GAMETIME [7, 6] for details. Experimental results indicate that in practice GAMETIME can accurately predict not only the worst-case path (and thus WCET) but also the distribution of execution times of a task from various starting environment states.

4 Looking Ahead

GAMETIME is only a first step and much remains to be done in the area of quantitative verification of software. First, the GAMETIME approach can be further refined to strengthen the theoretical guarantees and improve scalability, e.g., through compositional reasoning. Second, we need to understand what kind of support from platform designers (e.g., [1]) can make the quantitative verification problem easier. Third, many other quantitative verification problems, such as verifying bounds on energy consumption, remain to be fully explored. Finally, some of the fundamental ideas behind GAMETIME, such as the generation of feasible basis paths and the combination of inductive and deductive reasoning apply more generally beyond the area of cyber-physical systems.

Acknowledgments. This work was supported in part by NSF grants CNS-0644436, CNS-0627734, and CNS-1035672, an Alfred P. Sloan Research Fellowship, and the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

References

1. S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *Design Automation Conference (DAC)*, pages 264–265, 2007.
2. E. A. Lee. Computing foundations and practice for cyber-physical systems: A preliminary report. Technical Report UCB/EECS-2007-72, UC Berkeley, May 2007.
3. Y.-T. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic, 1999.
4. Reinhard Wilhelm et al. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 2007.
5. S. A. Seshia and J. Kotker. GameTime: A toolkit for timing analysis of software. In *Proc. Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, 2011.
6. S. A. Seshia and A. Rakhlin. Quantitative analysis of systems using game-theoretic learning. *ACM Transactions on Embedded Computing Systems (TECS)*. To appear.
7. S. A. Seshia and A. Rakhlin. Game-theoretic timing analysis. In *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 575–582, 2008.