Improving Timing Analysis for Matlab Simulink/Stateflow

Lili Tan, Björn Wachter, Philipp Lucas, Reinhard Wilhelm*

Universität des Saarlandes, Saarbrücken, Germany
{lili,bwachter,phlucas,wilhelm}@cs.uni-sb.de

1 Introduction

Control software in embedded hard real-time systems is subject to stringent timing constraints. To compute the required safe upper bounds on its worst-case execution-time (WCET), static timing analysis is used in industry [1].

Today control software is predominantly developed with model-based design tools such as Matlab Simulink/Stateflow. However, current timing tools lose precision as they consider infeasible executions, e.g., changes between operating modes not admissible in the model. These tools analyze compiled executables where information about the feasibility of executions is hard to derive. We propose systematic methods that make model information available to timing analysis and present promising results with Simulink/Stateflow models.

Static Timing Analysis. Static timing analysis [2] uses abstract interpretation [3] to derive program properties that hold for all executions. A classical static analysis is interval analysis, which determines, for each variable, a range of values for each program point which contains *all* the values of the variable in any program execution. The ranges are guaranteed to be safe, i.e., they can be used to exclude division by zero and array-out-of-bounds accesses at compile time. More generally, static analysis computes provably safe approximations of program states.

Static timing analysis determines execution time bounds for programs. These bounds must be safe, i.e., they must not underestimate the execution time. They should also be tight to avoid unnecessary safety margins.

The established methodology splits the problem into different phases. The input to the analysis is a compiled executable of the program. The first phase reconstructs from the executable a control-flow graph (CFG) over basic blocks. In the next phase, a variation of interval analysis, called value analysis, determines the contents of registers and memory locations. Then a micro-architectural analysis computes execution-time bounds for basic blocks. It accounts for the tremendous hardware-induced execution-time variability: depending on whether a memory access causes a cache hit or a cache miss, the execution time of an instruction may differ by two orders of magnitude. Therefore complex, processor-specific architectural features like cache and pipeline effects are considered [4]. In the final phase, path analysis determines a safe estimate of the WCET. First an

* Supported by ITEA 2 project 06042, EU-FP7 Grant 216008 and SFB/TR 14.

ILP generator models the control flow the program as an integer linear program. Each ILP variable corresponds to the traversal count of a basic block. The value of the objective function in the solution is the predicted execution time bound.

The CFG also describes infeasible program executions if *conditions* are not interpreted. Consider the C code `if(a>0) x=1; else x=2; if(a==0) a=x;`. Conditions `a>0` and `a==0` are clearly *correlated*, more specifically, they mutually exclude each other. Although the control-flow graph contains the path from `x=1` to `a=x`, this is not a feasible program execution. In general, we call a control-flow path an *infeasible path* if it does not correspond to any program execution (this notion is distinct from dead code).

To make path analysis more precise, so-called *flow constraints* can be added to the ILP that eliminate infeasible paths. A salient point of our work is that such constraints can be systematically derived from model information.

Matlab Models and Generated Code. Matlab Simulink/Stateflow is a hierarchical modeling language for control software with a sequential, imperative semantics. The underlying methodology is to design control computation within Simulink and control logic within Stateflow. Simulink offers building blocks for proportional, integral and differential (PID) control computations and estimations, e.g., filters, look-up tables, and arithmetic operators. Stateflow is an automata specification language that can be used to express transitions between different operating modes of the system. Blocks communicate with each other via signals and receive external inputs from the environment.

For deployment, code generators synthesize production C code, in which the internal states of Stateflow and Simulink blocks are encoded by state variables. Signals and internal inputs also map to C variables. The implementation of blocks can be traced in the source code. However this mapping depends on characteristics of different code generators.

2 Model-aware Timing Analysis

In real-time systems, the different tasks run periodically and are triggered by a scheduler. These tasks are commonly implemented with model-based tools like Matlab. A periodic run corresponds to one execution of the Matlab model where inputs are received, the internal state is updated, and outputs are produced. Timing analysis has to determine an execution time bound that is safe for *each* run. It is impossible in practice to know the worst-case inputs or the worst-case internal state, hence the analysis has to cover all possibilities for each run.

To ensure safety, the analysis must not assume that the value of an external input variable remains constant between definition and use, i.e., the variable is 'volatile' in C terminology. For the internal state, timing analysis has to assume all possibilities at task entry, i.e., for a state variable, assume all potential states. Thus, both input and state must be treated specially to obtain a safe execution time bound. In Matlab-generated code, input and state variables can be identified syntactically. This enables an automatic solution that guarantees safe bounds. In the remainder of the section, our goal is to make these bounds tighter.

We investigate where precision is lost due to infeasible paths. To this end, we focus on typical patterns at the level of the model that lead to infeasible paths. As a running example, we consider the fuel-rate controller which is a Matlab demo model that contains typical features of embedded controllers. The controller estimates airflow rate, and calculates the fuel injection rate based on PID control principle.

We analyzed the controller with `aiT` WCET Analyzer, the static timing analysis tool [2] of AbsInt [5]. `aiT` produces a worst-case path to explain the execution time bound it has computed. Without providing flow constraints, the execution time is over-approximated and the computed worst-case path is infeasible, since static timing analysis is not aware of certain dependencies in the model.

For example, like any control software, the fuel-rate controller has operating modes and signals that conditionally exclude each other. Depending on the current mode, signals, and their logical combinations, different look-up tables or computations are triggered. As discussed in the introduction, the timing analyzer generally does not interpret conditions. Hence it has to take the longer branch of a conditional, even if execution history of the path does not admit so. As a result, the worst-case path spuriously 'switches' between operating modes.

For illustration, we consider such spurious resolutions of conditions on the worst-case path. Some resemble the infeasible-path example in the introduction, e.g., they involve conditions like `mode==LOW` and `mode==RICH`. Other conditions are more involved. For example, condition `O2_fail==0 && mode==LOW` checks if the oxygen sensor is valid and the system is in operating mode `LOW`, while condition `pressure_fail==1` checks if the pressure sensor has failed. These conditions do not have shared variables, and, simply by looking at the expressions, they seem not to be related. Yet there is a relation entailed by the model: the conditions are, in fact, mutually exclusive. The conditions are used in a Simulink block, while the variables `mode`, `O2_fail`, `pressure_fail` are set by a Stateflow automaton. However, the Stateflow automaton would not set `mode` to `LOW` if any sensor had sent a failure signal. Such *entailed relations* need to be derived by analyzing the model semantics. In the source code or executable, dependencies are more implicit and even harder to track than in the model. In the following, we show how to construct flow constraints from the model to achieve a more precise timing analysis.

Trigger Conditions. We aim at conditions that determine whether a piece of the model is executed. These conditions on external inputs, internal signals (e.g., mode variables), and states guard signal transformation and control computation. Simulink/Stateflow express this by conditional blocks, similar to conditionals in C, e.g., triggered and enabled subsystems, guarded transitions in Stateflow and switch-blocks. We uniformly refer to the conditions as *trigger conditions*.

Flow Constraints from Definition-Use Dependencies. We formulate flow constraints that relate a definition, e.g., a mode variable, and uses of that variable. Certain definitions always make a trigger condition false. Trivially, a program execution cannot pass through such a condition *and* the branch guarded by the

trigger condition. This can be expressed by flow constraints. One example for such constraints in the fuel-rate controller are signals that indicate a failure of a sensor. These signals are set in a Stateflow block and are used in a Simulink block to trigger the evaluation of a lookup table.

Flow Constraints from Correlations between Trigger Conditions. Relations between trigger conditions can be formulated as flow constraints, e.g., independent, equivalence, implication, antivalence, and exhaustion can be expressed. To be effective, entailed relations need to be considered. The analysis of entailed relations requires information about deep semantic properties of Stateflow and Simulink blocks. To this end, we anticipate that relational abstract domains from static analysis may be helpful.

Other relations could be derived purely from Simulink. This includes the common case of a choice between two implementations of an algorithm with directly inverse trigger conditions.

Significant Branches. Eliminating infeasible paths does not per se improve precision. For example, if branches of conditionals have approximately the same execution time, there can be little gain in precision. Therefore, we focus on significant unbalanced branches when giving flow constraints. In our running examples, the invocations of look-up tables and mode-dependent discrete filters give rise to such branches.

Relative to Stateflow, the Simulink blocks typically dominate the execution time, while Stateflow blocks themselves contribute little to the overall execution time. This is because control logic computations consist of conditionals and assignments, while the expensive computations are often in the Simulink part, e.g., lookup tables and discrete filters for estimation and PID control. Thus determination of infeasible paths pays off more in the Simulink part than in Stateflow.

Experimental Results. We used `aiT` for our experiments. For the fuel-rate controller, we have manually applied the described derivation method for flow constraints. Flow constraints from definition-use dependencies alone reduced the execution time bound by 4%. Adding both kinds of flow constraints yields an overall reduction by 19% and a feasible worst-case path. If we compute an execution-time bound for each operating mode, we achieve a reduction from 20% to 48% per operating mode.

3 Related Work

Previous work on flow constraints focused on the executable [6], or C level. In [7], the authors consider timing analysis of code synthesized from Esterel. They identify flow constraints to eliminate feasible paths. The principal ideas concerning the two kinds of flow constraints are related, however Esterel is significantly different from Matlab Simulink, e.g., Esterel does not have automata as a language feature. Hence rules to derive flow constraints differ significantly.

[8] describes early work on timing analysis for Simulink models *without* Stateflow. Model information like loop bounds is passed to the underlying timing analysis tool. They modified the code generator and used their own (uncertified) compiler. Their timing analysis tool lacks value analysis [9] and thus does not discover loop bounds which `aiT` derives from the executable alone. Integrations of `aiT` with ASCET and SCADE are described in [10] and [11]. They pass model information to `aiT`, e.g., variable ranges and loop bounds. Unlike this paper, [8, 10, 11] mainly focus on other aspects than precision.

4 Conclusion

Initial results the benefit of model information in terms of automation and precision of WCET analysis. We propose model-based generation of flow constraints and have evaluated our method using the industrial tool `aiT`. Initial results with the fuel-rate controller are promising. While definition-use flow constraints are relatively easy to apply, relations between trigger conditions are more difficult to automate due to entailed relations. In future work, we will automate the generation of flow constraints and apply our approach to industrial examples.

References

1. Thesing, S., Souyris, J., Heckmann, R., Randimbivololona, F., Langenbach, M., Wilhelm, R., Ferdinand, C.: An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software Systems. In: Proceedings of DSN. (2003)
2. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: EMSOFT. Volume 2211 of LNCS. (2001) 469 –485
3. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL77, Los Angeles, California (1977) 238–252
4. Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R.: The influence of processor architecture on the design and the results of WCET tools. Proceedings of the IEEE **91** (2003) 1038–1054
5. AbsInt Angewandte Informatik GmbH: http://www.absint.com/
6. Stein, I., Martin, F.: Analysis of path exclusion at the machine code level. In: Proceedings of WCET. (2007)
7. Ju, L., Huynh, B.K., Roychoudhury, A., Chakraborty, S.: Performance debugging of Esterel specifications. In: CODES+ISSS. (2008) 173–178
8. Kirner, R., Lang, R., Freiberger, G., Puschner, P.: Fully automatic worst-case execution time analysis for Matlab/Simulink models. In: ECRTS. (2002) 31–40
9. Tan, L.: The worst-case execution time tool challenge 2006. International Journal on Software Tools for Technology Transfer (STTT) **11** (2009) 133 – 152
10. Ferdinand, C., Heckmann, R., Wolff, H.J., Renz, C., Parshin, O., Wilhelm, R.: Towards model-driven development of hard real-time systems. In: Proceedings of ASWSD. (2006) 145–160
11. Ferdinand, C., Heckmann, R., Sergent, T.L., Lopes, D., Martin, B., Fornari, X., Martin, F.: Combining a high-level design tool for safety-critical systems with a tool for WCET analysis on executables. In: Proceedings of ERTS. (2008)