SATIrE within ALL-TIMES: Improving Timing Technology with Source Code Analysis

Gergö Barany

Institute of Computer Languages, Vienna University of Technology

Abstract. We present the SATIrE source-to-source analysis framework within the context of ALL-TIMES, a European research and development project aimed at improving and integrating existing tools in the area of timing analysis. Within the project, SATIrE contributes by performing source-level static analysis on C programs and exporting its results for other tools to use.
This work gives an overview of SATIrE and how its analyses may improve timing analysis results obtained by other tools. We discuss SATIrE's efficient and powerful context-sensitive unification-based points-to analysis and its value interval analysis. We explain how SATIrE's integration with other analysis tools handles issues such as combination of source-level with binary-level analysis and communication of views of function call contexts between tools.

1 Introduction

With safety-critical embedded real-time systems controlling automobiles and airplanes, timing errors can have disastrous consequences. It is therefore very important to ensure that any hard deadlines specified for parts of the system can be met under all possible circumstances. Timing analysis aims to predict worst-case timing behavior in order to give feedback to developers and provide information to validation processes.
There are many forms of timing analysis, each with a number of advantages and disadvantages. Static analysis of source code benefits from being aware of symbolic variable names, data types, and program structure. However, actual *timing* results cannot be derived from the source code alone, since it does not uniquely determine the actual machine instructions that will be executed on the target platform. Hence, static analysis of the binary is needed to derive worst-case timings for code snippets. This level of analysis can make use of knowledge about actual machine code in conjunction with machine parameters such as instruction timings and predicted worst-case cache and pipelining behavior.
In another dimension, dynamic analysis relies on running the system to collect *observed* timings and other system characteristics, given certain inputs which are expected to cover worst-case circumstances. Dynamic analysis often results in tighter timings than static analysis, but it typically lacks a guarantee that the tested cases covered the worst-case behavior.

The goal of the ALL-TIMES project is to integrate various European tools which cover different points in this space of possible timing analysis approaches. This paper describes a part of the overall effort in ALL-TIMES, focusing on the SATIrE system and its connections to other tools within the project. The following section describes the SATIrE framework, the analyses it provides, and ways to extend it with further analyses; Section 3 discusses how SATIrE supports other tools in the ALL-TIMES project by exporting analysis information. Section 6 concludes.

2 The SATIrE System

SATIrE (Static Analysis Tool Integration Engine)[1] is a framework for integrating various static analysis and source code manipulation tools. Its focus is on programs written in the C programming language. SATIrE has been under development at Vienna University of Technology since late 2004.

2.1 Framework

SATIrE allows users to build tools that analyze or transform C programs. To this end, it provides interfaces to the ROSE[2] source-to-source transformation system; ROSE includes the C and C++ frontend by Edison Design Group and provides an object-oriented abstract syntax tree (AST) for manipulation. As an alternative frontend, SATIrE recently acquired bindings to clang[3]; clang's output can be translated by SATIrE into a ROSE AST.

Given the AST representing a program, there are several ways of analyzing and manipulating it. ROSE includes a number of analyses and transformations, including a loop optimizer that can perform common operations such as loop unrolling. To allow data-flow analysis on C programs, SATIrE includes a component that builds an interprocedural control-flow graph (ICFG) and bindings to the Program Analyzer Generator (PAG)[4] which generates data-flow analyzers from functional specifications. SATIrE also includes tools to export ASTs as Prolog terms and vice versa, to enable program analysis and transformation using the Prolog programming language.

ASTs that have been transformed or annotated with analysis results (in the form of comments or `#pragma` statements) can then be unparsed to C source code for further processing with other tools or compilers. The overall architecture of SATIrE is shown in Figure 1.

[1] `http://www.complang.tuwien.ac.at/satire/`
[2] `http://www.rosecompiler.org`
[3] `http://clang.llvm.org`
[4] `http://www.absint.de/pag/`

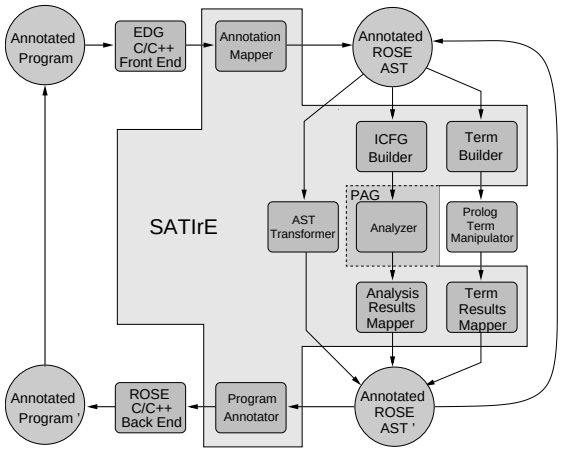

Fig. 1. Architecture of the SATIrE program analysis framework. The central program representation is the possibly annotated AST provided by ROSE. Various toolchains can work on this representation, possibly with intermediate transformations. The final result can be unparsed to annotated source code.

2.2 SATIrE Analyses

Besides providing interfaces for user-defined analyzers, SATIrE also includes a number of predefined analysis stages. The most important ones are the points-to analyzer, the value interval analyzer, and the loop bounds analyzer.

Points-To Analysis SATIrE's points-to analysis is a flow-insensitive unification-based analysis inspired by Steensgaard [Ste96]. The basic analysis performs a single pass over the program, assigning an abstract 'location' to each program variable, function, or dynamic memory allocation site. The effects of pointer assignments are modeled using points-to edges between locations. Each location is constrained to have at most one outgoing points-to edge; if some location might point to two or more different locations, those locations are merged into a new combined location.
This merging ensures that the analysis can be implemented in almost-linear time using a fast Union/Find data structure [Tar75]. However, it is also a source of imprecision as it may introduce spurious points-to relations that cannot be realized in any actual run of the program.
The basic algorithm suffers from the fact that it is context-insensitive; if a function that receives a pointer argument is called at several different sites, the analysis will merge all the objects that may be pointed to at any call site. Since passing and returning pointers is very common usage

in C, this can lead to considerable imprecision. Much of the lost precision can be regained by making the analysis context-sensitive: Based on some notion of interprocedural context, each function is analyzed multiple times. Argument and return-value locations of the function contexts are linked according to the calling relations between the contexts. This cloning approach is similar Lattner et al.'s context-sensitive points-to analysis [LLA07]. Currently, SATIrE uses context information derived by PAG, which can compute call strings of any bounded or (for non-recursive programs) arbitrary length.
Any external functions called from the program must have summaries available to the points-to analysis. Otherwise, they are treated conservatively as calling an unknown function that may introduce arbitrary aliases to global variables. The current implementation provides fully polymorphic summaries for part of the C standard library. Summaries are currently built into the analyzer, but an external annotation language for functions is under consideration.

Value Interval Analysis The interval analysis (sometimes called 'value range analysis') integrated in SATIrE derives possible values of integer variables in C programs. Each variable is associated with a pair of numbers representing an interval of values, which may be open on either side (lower bound of $-\infty$ or upper bound of ∞). The analysis is flow-sensitive, meaning that analysis information specific to every program point is derived. If at some point a variable is associated with an interval with bounds $[a, b]$, this means that at that point, the variable's value is definitely somewhere between a and b. This analysis is conservative, as in many cases not all values between a and b can actually be realized in executions of the program. However, all possible actual values are covered by the intervals derived by the analysis. The interval analysis is implemented as an abstract interpretation [CC77]. The declarative analysis specification is translated to an executable program using PAG.
Constant values in interval bounds typically come from direct assignment of a constant to a variable, such as initializing a loop counter to 0, or from comparison of a variable with a constant. For instance, a loop condition `i <= 10` gives an upper bound for the variable on the path entering the loop, and a lower bound for the path leaving the loop. In certain cases, the analysis can also make use of `assert` statements in the program that were inserted by programmers with domain knowledge, or by some other program analysis/transformation. The information in a statement like `assert(x >= 0 && x <= 10);` can be used by the interval analysis to infer that at the program point following the statement, the value of variable `x` must be in the range $[0, 10]$, regardless of what was known about its value before.
The analysis is integrated with SATIrE's points-to analysis. Integer assignments or reads through pointers can therefore be resolved to sets of possibly referenced variables. This can lead to much more precise results than other approaches, which cannot handle pointers and must make very conservative assumptions. In SATIrE's interval analysis, if only one variable can be referenced, its associated interval can be used or updated

as for direct reads or assignments; otherwise, the intervals for all concerned variables are read or updated in a conservative way that never underestimates the actual possible ranges of values. Similarly, an array variable is treated like a set of variables, and assignments to array elements do not replace but rather extend the interval representing all possible values stored in the array ('weak update').
The interval analysis is inter-procedural, i. e., intervals associated with argument expressions of function calls are propagated into the corresponding functions. Using facilities provided by PAG, the interval analysis can be used in a context-sensitive way with arbitrarily long call strings.

Loop Bounds Analysis SATIrE includes a component which computes loop bounds for loops based on iteration variables [PKST08]. It uses results of the interval analysis and structural information about the program to build a set of inequalities, which are solved to yield constraints on the numbers of loop iterations.
The loop analyzer constructs a set of inequalities for certain loops. Essentially, it looks for loops preceded by the initialization of an iteration variable, an exit condition consisting of an inequality involving the variable (or a set of such inequalities connected by 'logical or' operators), and exactly one increment or decrement of the variable inside the loop with a bounded step size. Experience shows that most loops in embedded systems software are of this form, so the analysis is widely applicable in the domain covered by the ALL-TIMES project.
Assuming the loop variable is i, the initialization expression is *Init*, the test expression is $i \leq \mathit{Test}$, and the step size *Step* is known to be positive, the following equalities and inequalities are generated:

$$i \geq \mathit{Init} \qquad i \leq \mathit{Test} \qquad (i - \mathit{Init}) \bmod \mathit{Step} = 0$$

The number of distinct solutions of this system is just the maximum number of possible iterations of the loop. An external constraint solver is used to calculate the number of solutions, which is often considerably more efficient than simply enumerating the solutions. Currently, SATIrE uses the *clpfd* solver distributed with SWI-Prolog[5] to calculate these loop bounds.
Applying the basic analysis recursively from outer to inner loops, nested counting loops can be analyzed as well. The loop bound for each inner loop is given in terms of the scope containing the outer loop. In particular, this means that triangular loops are not overestimated.

3 Integration in ALL-TIMES

This section describes the ALL-TIMES project and details SATIrE's involvement in the project.

[5] `http://www.swi-prolog.org`

3.1 ALL-TIMES Project Partners

ALL-TIMES comprises a total of six partners, four commercial companies and two university groups. Each partner contributes an existing tool with a specific approach to timing analysis and corresponding strengths and weaknesses. The partners have identified a number of valuable integrations between tools to combine and exchange various kinds of analysis data in order to obtain better results.
The partners and their tools are:

AbsInt Angewandte Informatik GmbH[6] develops the **aiT** family of WCET analyzer tools. aiT performs static analysis directly on the actual application binary. Using abstract interpretation, aiT derives possible value ranges of registers and memory locations; it also computes upper bounds on WCET of basic blocks, taking cache and pipelining effects into account. Using integer linear programming (ILP), aiT then derives a worst-case program path and the corresponding WCET bound from the basic block estimates.

Gliwa GmbH[7] is the developer of **debugGURU**, a dynamic analysis framework. It uses instrumentation of the target application to collect timing information at run-time. Using plugins, debugGURU can collect various kinds of information, including execution times and memory usage. The collected information can be communicated to host PCs using industry-standard bus systems.

Symtavision GmbH[8] provides **SymTA/S**, a system-level timing and scheduling analysis tool. SymTA/S works with a high-level abstract model of the application in terms of program tasks and resources such as CPUs and buses. It calculates resource loads, worst-case response times for tasks and worst-case transmission times for messages.

Rapita Systems Ltd[9] produces the **RapiTime** toolkit for dynamic analysis. RapiTime instruments the application with measurement code and uses the measurement to profile performance, provide code coverage information, and perform WCET analysis.

Mälardalen University's WCET group[10] uses its **SWEET** tool for timing analysis. Its flow analysis is based on abstract execution and can derive complex flow constraints relating execution frequencies for different points in the program. SWEET analyzes programs in the ALF representation [GEL+09], which is designed to be generated either from source code or from a binary.

Vienna University of Technology, compilers and languages group[11], contributes **SATIrE**, the framework for source-based analysis and transformation of C programs described in Section 2.

---

[6] `http://www.absint.com`
[7] `http://www.gliwa.com`
[8] `http://www.symtavision.com`
[9] `http://www.rapitasystems.com`
[10] `http://www.mrtc.mdh.se/projects/wcet/`
[11] `http://www.complang.tuwien.ac.at`

4 SATIrE's Connections in ALL-TIMES

This section explains the tool integrations in the ALL-TIMES project that SATIrE is involved with.

4.1 Integration: SATIrE-RapiTime

The integration of SATIrE with RapiTime involves communication of analysis information from SATIrE to RapiTime. The main goal is to provide information about function pointers, while information regarding loop bounds may also be useful to RapiTime.

As RapiTime uses dynamic analysis to gather information at run-time, one cannot always be sure that all possible executions of certain parts of the code have been covered by its analysis. RapiTime therefore provides the possibility for users to annotate the program's source code with high-level knowledge about issues such as points-to relations or flow constraints. Within the ALL-TIMES project, SATIrE will use static analysis to compute some of the information that would otherwise be provided manually. This saves users from part of the tedious and error-prone task of program annotation.

In order to compute a worst-case timing for a function call, RapiTime must know all the possible functions that may be called at that site (in a certain context). Since embedded systems codes often contain indirect calls through function pointers, this information is typically not immediately available. During execution of the system, the code annotated by RapiTime can record all *observed* functions called from a certain site, but as noted above, it may not always be sure that these were all the *possible* call targets for that call site. Without this information, it must make a conservative approximation or reject the program.

This is where analysis information from SATIrE can aid RapiTime in deriving tight WCETs: Its points-to analysis can give static information about possible call targets at each call site. The automatic analysis is much faster and more reliable than manual annotations; this is particularly true for context-sensitive annotations. Thus SATIrE's information can tell RapiTime whether it has observed all possible call targets during its tests, or which other possible targets to take into account for its WCET calculation.

Similarly, RapiTime may observe certain numbers of iterations for loops in the application. SATIrE's static analysis of loop bounds may confirm that the observed iterations are indeed the worst case, or provide information for appropriate computation of a guaranteed time bound.

While RapiTime allows users to annotate source code with `#pragma` statements containing analysis information, SATIrE's analysis information will be communicated using an external file format. Source code annotations are very natural for manual annotations, but for automated exchange of information an external format separated from the source code may offer more flexibility.

4.2 Integration: SATIrE-aiT

For the integration with aiT, SATIrE will provide analysis information on points-to relations, unreachable code, and its view of interprocedural contexts, which differs from aiT's.

The static analysis performed by aiT on the application binary takes a purely numeric view of data, including pointers: Pointers are treated as numeric addresses, and aiT's value analysis abstracts their possible values as numeric intervals. This is especially problematic for global entities such as global variables or functions, where a symbolic analysis based on names can be much more exact that pure numeric analysis.

Consider, for instance, a function pointer which may point to two functions `f` and `g`. If the compiler's code layout is such that these functions are not adjacent in the binary, a numeric analysis will determine that the pointer may point to the address of `f` or the address of `g` or any other function that lies between them in the code. Those functions in-between, however, are not actually feasible targets but rather just artifacts of a numeric analysis. Since SATIrE's points-to analysis works on names rather than addresses (which do not even exist on the source level), it is not susceptible to this kind of spurious result. The same reasoning applies to the treatment of pointers to data, which is why SATIrE may also derive much more precise analysis information for pointers to global buffers or stack variables.

As noted in Section 2.2, SATIrE's interval analysis is integrated with its pointer analysis. Thus, in certain cases, the more precise pointer information in SATIrE can be used to derive variable values that aiT is not able to compute. Such values can then be used to identify branch conditions that are always true or always false in certain contexts, and this more precise flow information can result in a better WCET estimation.

Other kinds of information SATIrE exports to aiT are points-to relations for data pointers and possible targets of calls through function pointers.

An important issue in integrating SATIrE and aiT is the fact that these tools have different notions of interprocedural contexts. In aiT's view of the program, loops are transformed into special tail-recursive procedures; a jump back to the loop's head for another iteration corresponds to a tail call. This approach allows aiT to use call strings to model loop contexts, which means that it need not merge analysis information from the first N loop iterations with the information from later iterations. On the other hand, this handling of loops as calls means that direct exchange of call strings between the two tools is not possible.

To communicate context information to aiT, SATIrE uses aiT's concept of 'user-defined registers'. These are pseudo-registers that have no counterpart in the actual code and hardware, but rather only contain values at analysis time. Annotations to the code may refer to, and store, values in these user registers. aiT's value analysis propagates these values just like it propagates values of actual hardware registers and memory locations.

Context information from SATIrE is then communicated as follows: Each function in the program has an associated 'call site' user register. Each call site is associated with a unique numeric identifier. At each call site,

annotations from SATIrE instruct aiT to store the call site register for each possible target function with that site's ID. Within called functions, annotations may be qualified to hold only if the call site registers contain certain specific values. For instance, an annotation restricted to be true only if three registers contain certain call site identifiers essentially corresponds to an annotation that is specified to hold only given a certain three-element call string. The analogy to finite call strings only breaks down in the presence of recursion, in which case having just one register per function makes the analysis less precise—however, recursion is not a major concern in typical embedded control applications.
Information from SATIrE is communicated to aiT using source code annotations, or external annotations that refer to source code locations. aiT does not itself work with source code; rather, it relies on the compiler that generates the binary to also output debug information that relates source code positions to addresses in the executable.

4.3 Integration: SATIrE-SWEET

The integration of SWEET and SATIrE involves various issues. First, as SWEET works on programs in the ALF representation [GEL+09], it needs translators to ALF in order to be able to analyze binaries or source code. One part of the connection between SATIrE and SWEET is therefore a C-to-ALF compiler called melmac[12]. This is a fairly regular C compiler backend, but its tight integration with SATIrE also allows it to output analysis results and meta-information that is useful for the other aspects of the connection.
Thus for the second part of the connection, melmac also outputs information mapping ALF code positions (identified by jump labels) to SATIrE's internal position identifiers as well as source code locations, and information on the call strings used by SATIrE. Using this information, SATIrE's context-sensitive analysis results regarding points-to information and value intervals can also be communicated to SWEET. In contrast to the other connections described above, SWEET and SATIrE have similar notions of program objects (ALF allows named, scoped variables like C does). Thus SATIrE's analysis information referring to program variables and pointer relations is directly useful for SWEET. The tight correspondences between program positions as well as variables allow SATIrE to exchange flow-sensitive interval information with SWEET, which is not the case for the other connections. This can ease the implementation burden on SWEET's developers, who at the time of writing do not have a context-sensitive points-to analysis.
The third and final part of this connection involves communication of analysis information from SWEET to SATIrE. SWEET's strength lies in deriving complex flow facts using abstract execution; these facts identify mutually exclusive program points or give constraints on the relative number of executions of different program points. Currently, SATIrE itself cannot make use of this kind of detailed flow information; however, it can output the flow facts for use by aiT. Thus SATIrE can play the

[12] `http://www.complang.tuwien.ac.at/gergo/melmac/`

role of a translator in a connection involving two other tools that might otherwise not be able to exchange information.

5 Related Work

Gustafsson et al. [GLS+08] give a more detailed overview of the ALL-TIMES project and the tools involved. Schordan [Sch08] presents the SATIrE system in more detail and considers some challenges of annotating source code with analysis information.
There does not appear to be much previous work that deals specifically with integration of source code analysis and WCET calculation. The TuBound tool [PSK08] is an exception: TuBound integrates a source-based tool which derives loop bounds and adds corresponding source code annotations, and a C compiler which includes a component for WCET computation that can make use of these annotations. The source-based part of TuBound is built on SATIrE and the same analyses that SATIrE contributes to ALL-TIMES; however, TuBound's integration of a WCET-aware compiler is different from any of the work reported here.

6 Conclusions

We have presented the SATIrE program analysis framework and described its role within ALL-TIMES, a project aimed at integrating European timing analysis tools. In this project, SATIrE exchanges information with tools that employ a wide range of static and dynamic analysis techniques involving different levels of source, binary and intermediate code. Each connection allows SATIrE to aid the other tools in specific ways by providing valuable information derived using static analysis of source code. The benefits of this integrated approach are expected to validate the basic premise of the ALL-TIMES project, which is that combination of different approaches can yield much better results than each approach in isolation.

References

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[GEL+09] Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg, and Linus Källberg. ALF – a language for WCET flow analysis. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, June 2009.

[GLS+08] Jan Gustafsson, Björn Lisper, Markus Schordan, Christian Ferdinand, Marek Jersak, and Guillem Bernat. ALL-TIMES - a European project on integrating timing technology. In *Proc. Third International Symposium on Leveraging Applications of Formal Methods (ISOLA'08)*, pages 445–459. Springer, October 2008.

[LLA07] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 278–289, New York, NY, USA, 2007. ACM.

[PKST08] Adrian Prantl, Jens Knoop, Markus Schordan, and Markus Triska. Constraint solving for high-level WCET analysis. In *The 18th Workshop on Logic-based methods in Programming Environments (WLPE 2008)*, pages 77–89, Udine, Italy, December 2008.

[PSK08] Adrian Prantl, Markus Schordan, and Jens Knoop. TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis. In *8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, pages 141–148, Prague, Czech Republic, 2008. Österreichische Computer Gesellschaft.

[Sch08] Markus Schordan. Source-to-source analysis with SATIrE - an example revisited. In *Scalable Program Analysis*, number 08161 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

Acknowledgements

This work is supported by the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No. 215068 funded by the 7th EU R&D Framework Programme. See the project web site at `http://www.all-times.org` for more information about ALL-TIMES.