Analysing Switch-Case Code with Abstract Execution*

Niklas Holsti[1], Jan Gustafsson[2], Linus Källberg[2], and Björn Lisper[2]

1 **Tidorum Ltd.**
Helsinki, Finland
`niklas.holsti@tidorum.fi`

2 **School of Innovation Design and Engineering, Mälardalen University**
Västerås, Sweden
`{jan.gustafsson,linus.kallberg,bjorn.lisper}@mdh.se`

Abstract

Constructing the control-flow graph (CFG) of machine code is made difficult by dynamic transfers of control (DTC), where the address of the next instruction is computed at run-time. Switch-case statements make compilers generate a large variety of machine-code forms with DTC. Two analysis approaches are commonly used: pattern-matching methods identify predefined instruction patterns to extract the target addresses, while analytical methods try to compute the set of target addresses using a general value-analysis. We tested the abstract execution method of the SWEET tool as a value analysis for switch-case code. SWEET is here used as a plugin to the Bound-T tool: thus our work can also be seen as an experiment in modular tool design, where a general value-analysis tool is used to aid the CFG construction in a WCET analysis tool. We find that the abstract-execution analysis works at least as well as the switch-case analyses in Bound-T itself, which are mostly based on pattern-matching. However, there are still some weaknesses: the abstract domains available in SWEET are not well suited to representing sets of DTC target addresses, which are small but sparse and irregular. Also, in some cases the abstract-execution analysis fails because the used domain is not relational, that is, does not model arithmetic relationships between the values of different variables. Future work will be directed towards the design of abstract domains eliminating these weaknesses.

1998 ACM Subject Classification F3.2 Semantics of Programming Languages: Program Analysis

Keywords and phrases dynamic control flow, indexed branch, machine-code analysis, WCET analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2015.85

1 Introduction

Static analysis of the worst-case execution time (WCET) of a program usually begins by building the control-flow graphs (CFG) of all subprograms, and the call-graph connecting the subprograms. On the machine-code level, where most conventional WCET tools work, the tool has to find the possible successor instructions of each instruction under analysis. This is easy when the instruction defines its successors statically but hard for control-transfer instructions with dynamic target addresses, for example register-indirect jumps. Such *dynamic transfer of*

* This work was supported by the APARTS project in the FP7 Marie Curie IAPP programme (grant agreement no. 251413), and by COST Action IC1202: Timing Analysis On Code-Level (TACLe).

control (DTC) instructions often result from switch-case statements. Other reasons for DTC include function pointers, virtual function calls in object-oriented languages, and closures in functional languages.

This paper focuses on DTC from switch-case statements. This analysis is typically local and intra-procedural, while function-pointer analysis is a global value analysis and assumes that the CFGs of all procedures are already known. The switch-case statement in languages such as C or Ada is a very flexible control structure. The programmer can choose the type of the switch index, for example an 8-bit or a 32-bit number; whether the cases are numbered densely 1 .. n or are a sparse subset of a large range; whether each case is reached by a unique index value or by a set or range of values; and whether there is a default case or not. Compilers often generate quite different kinds of code to implement different kinds of switch-case statements. Moreover, hand-written assembly code (still often found in run-time libraries) can have quite tricky switch-like structures. Switch-case statements are also common in real programs and a failure to analyse even one of them can prevent the analysis of the whole program. Of course, the compiler knows all the targets of each switch-case DTC, and in principle could make this information available to a WCET analyzer, but in practice we cannot expect to get this information from all the industrial compilers, and especially not for manually coded assembler subprograms – even debugging information is sometimes unavailable for those.

On the surface, analysing a DTC instruction is just value-analysis: we need to know the possible run-time values of the variable or register that holds the final target address. However, in practice we have more requirements – the analysis of the DTC generated from a switch-case statement should:

- produce the precise value-set, because any over-estimation will introduce false execution paths, which can make a mess of the rest of the analysis,
- produce the sequence of instructions that leads to each case, so that later steps in the analysis can find an accurate WCET for each case,
- connect each case with the corresponding values of the switch index, again for use in later analysis steps (for example to find bounds for a loop that is nested in a case and depends on the switch index),
- apply uniformly to most types of code for most kinds of switch-case statements, and be robust to changes in the code as the compilers evolve or manually written code is revised. Adding new patterns or other special cases for every new version of every compiler for every supported processor is cumbersome and expensive.

The more general analysis-based methods have a larger chance of satisfying the last point, while more specialized pattern-based methods will suit the first three points better.

How can a value-analysis method meet these requirements, in particular a method using abstract interpretation? Firstly, for high precision a very suitable and accurate abstract domain must be used, and imprecision induced by merging abstract values must be limited. Secondly, because switch-case code is often optimized and quirky, the “smooth” and theoretically appealing abstract domains such as intervals or polyhedra may not work well. Thirdly, this code often uses memory-resident tables, so the analysis must model memory values as well as registers. However, these tables usually contain constants, so a model of indexed access to constant memory data may be enough, and mutable memory data need not be modelled.

Here, we apply the *abstract-execution* method of the SWEET tool to switch-case code, and compare it to the switch-case analyses in the Bound-T tool, which are in part pattern-based. This is also an experiment in modular tool design where a general value analysis tool is used as a plugin to a WCET analysis tool. Indeed, this scheme should also be applicable to other value- and WCET analysis tools if they are provided with suitable interfaces.

SWEET requires ALF code [5] as input, not machine code. Thus, we needed a front-end to translate machine-code into ALF. We implemented this function in Bound-T, which in the future will let users of Bound-T benefit also from other SWEET analysis results like loop bounds and other flow-facts. However, so far we have implemented the link between Bound-T and SWEET only for DTC analysis.

The rest of this paper is organized as follows. Section 2 introduces our tools and methods: abstract execution, SWEET, Bound-T, and the coupling of Bound-T and SWEET to use abstract execution for DTC analysis. Section 3 describes the test programs we used, in particular the machine-code form of their switch-case statements. Section 4 discusses the results of the analysis of the test programs, tracing the reasons for successes and failures. Section 5 briefly surveys some related work on analysing DTC. Section 6 draws some conclusions for future development of SWEET and Bound-T.

2 Tools and methods

2.1 SWEET and Abstract Execution

SWEET [10] was originally aimed at WCET analysis. The current version of SWEET focuses on value-analysis and control-flow analysis. SWEET analyses programs represented in the ALF language [5]. ALF is a textual language, similar to a compiler's intermediate representation, and is designed to represent code on both low- and high level (like machine code, or C) faithfully. An ALF program consists of global data declarations and functions. A function consists of local data declarations and a sequence of assignment statements interleaved with branch statements, where any statement can be labelled for use as branch target.

In addition to the conventional value analyses based on abstract interpretation with widening, SWEET provides *Abstract Execution* (AE) [6]. This can be seen as a very context-sensitive value analysis without widening, where different loop iterations are analysed separately. This makes the analysis resemble a symbolic execution where the program is executed with abstract states, in the abstract domain, rather than in the concrete domain. This yields a more precise value-analysis, but has the draw-back that it may not terminate.

During the AE, several abstract states may appear. When the AE reaches a control-flow join, it may or may not *merge* the abstract states from the incoming paths using the least upper bound operator in the abstract domain. The merging is controlled by several command-line options and can even be completely shut off. Disabling merging has the effect that the abstract states that provide the result of the value analysis are not merged, which effectively turns the abstract domain into its powerset domain. This makes it possible to obtain sparse value sets from the AE even when a regular abstract domain such as intervals is used. This matters because DTC target addresses often form sparse sets.

The current version of SWEET supports both intervals and *Circular Linear Progressions* (CLP) [13] as abstract domains. CLP combines the well-known intervals with *congruences* [4]: the latter can represent non-unit address strides, which is important for DTC analysis.

Our aim is to use AE, with no merging, to compute the possible values of the target addresses of the DTC instructions in the machine-language program under analysis. The question then arises how DTC instructions can be represented in an ALF program.

2.2 Dynamic control flow in ALF

ALF statements can have labels which have a symbolic part and a numeric offset part. Such *statement references* are first-class ALF values and can be computed dynamically by selecting a value for the symbolic part from those existing in the program, and/or by numerically computing a value for the offset part. The computed statement reference can then be used in a jump statement – an ALF DTC. SWEET's analysis can produce a safe (i.e. possibly over-estimated) set of possible DTC targets, as ALF statement references.

However, when the ALF code is generated from machine code, modelling a DTC with ALF statement references would require the ALF code to include all the ALF statements representing the instructions that may be the targets of the DTC. Therefore these instructions would have to be found and translated into ALF, before the ALF representation of the DTC could be subjected to analysis. In effect, this would require the ALF generator itself to resolve the DTC, before generating the ALF program.

The reader may ask, why not translate *all* of the machine code, as found in the code-memory image, into ALF statements? Then all possible DTC targets would be present in the ALF program. However, the presence of data intermixed with the code, and the existence of variable-length instructions and the consequent ambiguity of where instructions start, and the possible presence of more than one instruction set or encoding in the same program mean that this approach could generate a very large ALF program, much of which would be nonsense because it would not represent real instructions. Furthermore, an ALF program is syntactically divided into functions (subprograms), and an ALF DTC jump is permitted only within an ALF function, not across functions. A translation of the whole program into ALF would have to group the instructions into functions, which is impossible without resolving the DTCs in the machine code, or without some meta-information on function boundaries in the machine code. Similar reasons prompted Theiling's use of bottom-up CFG reconstruction [14] as opposed to the top-down, decode-all-the-image approch used by some earlier work.

Therefore, we allow ALF programs to be incomplete. We make the ALF program end at the point of an unresolved DTC, and analyse the incomplete program to discover the values of the numerical variables on which the DTC target depends. The analysis results are then used to compute the DTC target addresses. These addresses are passed back to Bound-T, which in turn generates a new ALF program that includes code reachable from the resolved DTC's. This iteration goes on until all DTC's are resolved – see Section 2.4.

2.3 Bound-T and its analyses of dynamic control flow

Bound-T is a WCET and stack-usage analyser using pure static analysis of linked, executable machine-code programs [11]. Bound-T's DTC analyses are always implemented or initiated by processor-specific and sometimes compiler-specific analysis procedures, triggered by specific code patterns. Still, the specific DTC analyses build on and use the general analyses in Bound-T. The DTC analyses fall into two groups:

- *Abstraction-based analyses.* Here, the target-specific part of the analysis models the DTC in a form that is amenable to one of the abstraction-based analyses in Bound-T: constant propagation, value-origin (*def–use*) analysis, and Presburger-arithmetic analysis. The latter is incidentally described in [8] and is based on the Omega Calculator [16].
- *Partial-evaluation analyses* [7]. Here, the target-specific part of the analysis is responsible for starting, stopping and parametrizing the partial evaluation.

The main limitation of the abstraction-based analyses is that the Presburger-arithmetic analysis cannot model dynamically addressed memories, only statically identified variables.

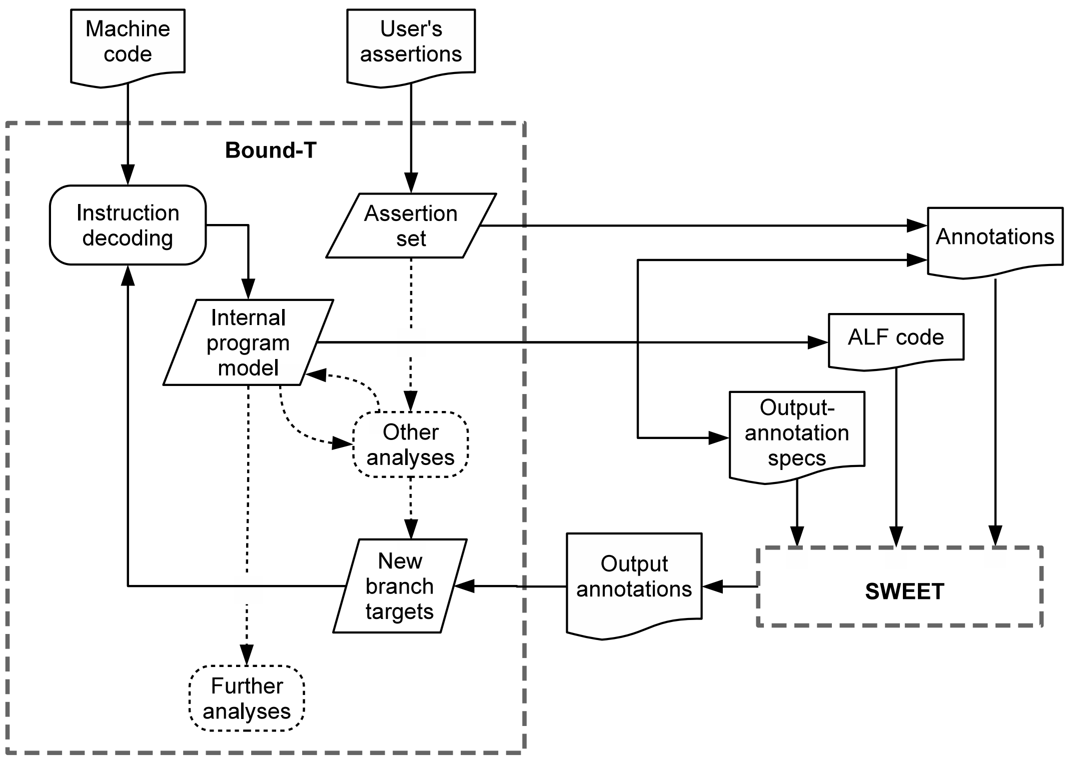

Figure 1 Overall analysis flow.

If the DTC involves a table of some form, the Presburger-arithmetic analysis must be focused on the addresses of the table elements, while the reading and interpretation of the data in the table must be done separely and depends on the DTC type.

The main limitation of the partial-evaluation analyses is the need to detect where and how the evaluation should be started – typically on entry to specific library functions ("switch handlers"). Some manual reverse engineering of these functions is needed, to understand how they access the "switch-tables" encoding the switch-case statements.

2.4 Coupling SWEET and Bound-T

To evaluate abstract execution for DTC analysis, we extended Bound-T to export its internal program model into ALF and to run SWEET on that ALF program, asking for AE using either the interval domain (which is sensitive to variable bit-widths and wrap-arounds) or the CLP domain, no merging, and output of the abstract values of the variables which determine the targets of the DTC's. (However, when merging is disabled, the values produced are sets of concrete values rather than abstract values.) Bound-T then reads the SWEET output, computes the corresponding DTC target addresses, and adds those instructions and their static successors to its internal program model, if necessary fetching and decoding more code from the executable under analysis. The extended program model may contain new DTC's. This process (ALF generation — SWEET analysis — completion of program model) is iterated until it stabilizes. Fig. 1 illustrates the data flow in this process. The iterative construction and extension of program models involving DTC was already implemented in Bound-T for its own DTC analyses, so essentially we only added SWEET as a further method of DTC analysis. For technical details see [9].

We implemented this coupling for the Atmel AVR 8-bit processor architecture, a widely used microcontroller family [1]. The AVR is a difficult subject for DTC analysis because it implements mostly 8-bit operations and no general $base + index$ addressing. The AVR has a Harvard architecture with separate instructions for reading data from code memory. Code addresses are 16 or 22 bits wide (depending on the AVR device) and are manipulated as pairs or triples of 8-bit data. Some of the 32 general 8-bit registers can be paired and used as 16-bit registers, but the set of 16-bit operations is quite limited. The AVR machine code generated for switch-case DTC is complex and varied. This makes pattern-matching analyses unreliable and favours semantically based analyses, such as AE.

3 Test programs

We selected 6 programs from the Bound-T test suite and added one new program to the suite, giving a total of 7 test programs (P1 – P7 below). The programs can be provided on request without IP restrictions. The goal was to include many different forms of switch-case code using DTC. Note that it is the machine-code form of the DTC that is important, not the source-code form. We did not use the well-known MRTC benchmarks because their switch-case statements produce simple or no DTC code with the compilers we have tried. The switch-case structures in our test programs are either extracted from real programs, or written to be similar to switch-case statements in real programs. The DTC machine codes are representative of Tidorum's experience with real programs and several different compilers. The chosen programs are small, because the prototype implementation of ALF export in Bound-T cannot yet handle complex parameter-passing confidently. However, analysis of switch-case code is for the most part local and intra-procedural so the program size is not important.

P1 A bottom-test loop, which contains a dense switch-case statement implemented by an indexed dynamic jump into a table which, in turn, contains static jumps to the code for each case. The loop counter is 8 bits, running from 15 to 19, and the cases are also numbered 15 to 19 with no default case.

P2 A dense switch-case statement with an 8-bit index, implemented by loading the 16-bit jump target address from a table in code memory, in two 8-bit parts, and executing an indirect jump to that address. The cases are numbered 0 to 9 plus a default case.

P3 A sparse switch-case statement with an 8-bit index, four cases plus a default, implemented by a switch-table and the corresponding switch-handler subprogram. This is a simplified, artificial switch-table structure from the example in [7].

P4 A sparse switch-case with an 8-bit index, four cases plus a default, implemented with the real switch-table structure (sparse variant) and real switch handler used in the IAR Systems C compiler.

P5 A dense switch-case statement with an 8-bit index, ten cases plus a default, implemented with the real switch-table structure (dense variant) and real switch handler used in the IAR Systems C compiler.

P6 A 16-case switch, implemented by an indexed jump into a dense table of jumps, in which the 4-bit index is assembled from two 2-bit pieces using "rotate" followed by "or". In effect, this is a 2-dimensional switch-case with two indices of 2 bits each. Jump-table elements are two addressing units long.

P7 A variant of P6 in which the "rotate" instruction is replaced by a "left shift" instruction and jump-table elements are one addressing unit long.

Table 1 Analysis results.

Program	*Bound-T result*	*SWEET (interval) result*	*SWEET (CLP) result*
P1	*Exact result*, but see note.	*Exact result.*	*Exact result.*
P2	*Fails.*	*Fails.*	*Fails.*
P3	*Fails.*	*Exact result.*	*Exact result.*
P4	*Exact result.*	*Exact result*, but see note.	*Exact result*, but see note.
P5	*Exact result.*	*Fails.*	*Fails.*
P6	*Fails.*	*Fails.*	*Exact result.*
P7	*Exact result.*	*Exact result.*	*Exact result.*

We know of some forms of switch-case code that are not represented in this test set: tables of *code offsets*, instead of code addresses, and address (or offset) tables accessed by *hashing* the case index, instead of using the case index itself (minus some lower bound) as an offset into the table. There are probably more strange forms lurking in the code jungle.

4 Analysis results

We compiled the test programs into AVR executables (choosing either the gcc or the IAR compiler, to generate the desired kind of DTC machine code) and analysed them with Bound-T alone (disabling the use of SWEET) and with the Bound-T + SWEET combination (disabling Bound-T's own DTC analyses) using either the interval domain or the CLP domain, with no merging over paths. For these small programs, no analysis lasts over 10 seconds on a MacBook Air with a 2.13 GHz Intel Core 2 Duo processor. Table 1 shows the results. For a detailed discussion of each success and failure, refer to [9].

For P1, Bound-T alone needs a manual assertion that the switch-case index is non-negative. The SWEET domains have better models of signedness and wrap-around.

For P2, Bound-T alone fails because the arithmetic analysis cannot model general addressable memories, and because a specific pattern for this "load-address-from-table" idiom is not implemented. The interval domain fails because lack of congruence information leads to over-estimation of the octet pointer into the address table, which causes such huge over-estimation of the DTC targets that Bound-T rejects the solution. The CLP domain gives the exact set of strided octet pointers into the address table, but the set of address values cannot be exactly merged into a single CLP value (such merging of table values cannot be disabled in SWEET). This causes significant over-estimation of the DTC targets, which in the third iteration leads to such an over-estimated CFG that SWEET's AE fails to terminate in a reasonable time, probably because a spurious eternal loop was introduced.

For P3, Bound-T alone fails because this switch handler is an artificial one for which Bound-T has no detector. Otherwise the partial evaluation method would work here.

For P4, SWEET with either domain gives the exact result for the DTC, but the WCET is overestimated. Bound-T's partial-evaluation method fully unrolls the search loop in the switch-handler, letting Bound-T find the exact worst-case path. The analysis with SWEET retains the loop and Bound-T then calculates the WCET under the normal assumption that all loop iterations use the worst-case path through the loop body. This WCET overestimation could be avoided either by transporting more flow-facts from SWEET to Bound-T, or by letting SWEET compute the WCET as a program variable [8].

For P5, the interval domain fails because it is not relational and does not include congruence. The CLP domain fails because it, too, is not relational, and because it does not see repeated additions of the same value (for example $x + x$) as equivalent to multiplication ($2x$) and therefore loses congruence information.

For P6, Bound-T alone fails because the carry-out from the "rotate" operation is not well modelled in the arithmetic analysis. SWEET with the interval domain fails because lack of congruence leads to over-estimation of the pointer into the jump table, leading to over-estimation of the DTC targets. However, in this contrived and simple case, the over-estimation only doubles the set of targets, so Bound-T accepts the result, leading to an apparently successful analysis but an over-estimated CFG and WCET. The CLP domain succeeds because the jump instructions in the jump table lie at regularly spaced addresses, which can be represented exactly in a single CLP value.

For P7, the exact result from Bound-T alone depends on some unsound assumptions in modelling left-shifts. SWEET succeeds with the interval domain because the unit stride of the jump-table elements makes congruence analysis unnecessary.

In total, Bound-T alone succeeds on 4 and fails on 3 programs. SWEET's analyses succeed on 5 and fail on 2 programs. If the analyses are combined, only P2 is left as a failure. Further patterns and special cases could easily be added to Bound-T to correct the failures, including P2, but that would just be one more step on the endless trail of special cases.

The results show that the abstract-execution method is very promising, working in essence as well as and in some cases better than the set of processor- and compiler-specific and manually constructed pattern-based methods in Bound-T. Comparing the two numerical abstract domains, the CLP domain succeeds in one case (P6) where the interval domain fails. We expected that CLP would yield a larger improvement, but its benefits are often masked by the direct translation from machine instruction semantics into ALF code, which hides congruence information when, for example, a C source instruction such as `y=2*x` is translated into two AVR instructions that do `y=x; y=y+x`. The Presburger-arithmetic analysis in Bound-T, being relational, is able to combine the two instructions and conclude that $y = 2x$. When Sen and Srikant introduced the CLP domain [13], they combined it with the domain of affine equalities, perhaps to counter this sort of problem.

Some of Bound-T's failures are due to poor modelling of signedness and wrap-arounds. The Presburger-arithmetic formalism considers all variables to be signed, unbounded integers. Wrap-around can be modelled with conditional (disjunctive) formulae, but if that is done systematically the time and space requirements explode. In contrast, SWEET's interval and CLP domains handle wrap-around gracefully (but not always exactly). However, because we disable merges in SWEET, it may also run into scalability problems for large programs.

The failure of SWEET on P5 is interesting. P5 computes the address of the table element for the DTC target address in three steps. Step 1 computes an intermediate value from the switch index. Step 2 compares the switch index to the range of case numbers. If the switch index is in range, step 3 computes the final address using the switch index and the intermediate value. Because the intermediate value is computed before the switch index is known to be in range, and because the domains are not relational, SWEET can put no useful bounds on the intermediate value, and so the final address is also unbounded.

5 Related work

Our work follows the "bottom-up" CFG reconstruction approach as also described by Theiling [14]. However, it seems that Theiling does not expect strong value-analysis to be used for switch-case code, saying that the "decoders may use pattern matching and code slicing to detect switch tables". Cifuentes and Van Emmerik [3] use slicing and symbolic expression substitution to simplify switch-case code so that it can be matched to a library of "normal forms", an advanced and flexible form of code-pattern matching. However, their experiments use wide-word, powerful processors of the SPARC and Pentium class, avoiding the complications of 8-bit microcontrollers such as the AVR.

The CodeSurfer/x86 tool described by Balakrishnan and Reps [2] has many similarities with SWEET and some with Bound-T. CodeSurfer/x86 analyses Intel-x86 machine code. The initial control-flow and call-graphs are generated by the commercial IDAPro disassembler [12], which may leave some switch-case DTCs unresolved. A value-analysis based on abstract interpretation, using a domain of intervals with strides (congruence), provides possible DTC targets for an iterative extension of the CFG, as in our method. However, CodeSurfer/x86 is at present limited to the Intel-x86 instruction set and some of the decisions made during the analysis seem specific to this architecture. The value-analysis uses only 32-bit quantities, and it is not clear how well it models unsigned computation. The basic model is signed.

Balakrishnan and Reps [2] agree with two of our conclusions: that a domain modelling non-unit strides is necessary for this analysis, and that non-relational domains can cause important loss of precision. However, as we have seen, quite many DTCs have sets of target addresses which cannot be represented exactly as strided intervals. When merging is disabled, AE can produce more precise results.

6 Summary and conclusions

Focusing on the analysis of switch-case code using DTC, we compared SWEET's AE (with no merging) to the set of special, pattern-based analyses in Bound-T, on seven programs expressed in Atmel AVR machine code. AE won, on average, but failed for some programs because the numerical domain we used (CLP) is not relational. Other failures are due to the inability of intervals, even with congruence, to model sparse integer sets precisely.

Tidorum Ltd aims to make the SWEET-based DTC analysis a standard feature of Bound-T. This will also let Bound-T use other SWEET results such as flow facts. The main challenge is to make Bound-T's data-memory model, and its translation to ALF, sound with respect to aliasing. We also found weaknesses in Bound-T's handling of signedness and wrap-around. Tidorum aims to improve this, but the problem is complex if the model should be relational (as the current Presburger-arithmetic model is).

This work was partly responsible for prompting the implementation of the CLP domain in SWEET. Future work on SWEET, to better support the analysis of DTC's, includes the design of light-weight relational domains that can capture enough of the dependencies between variable values to gain the necessary precision. It also seems interesting for this purpose to have an abstract domain that can represent sparse sets of integers up to some predefined size of the set.

A more general question is whether it is possible and desirable to perform abstract interpretation with different domains, for different variables or different parts of the program. The DTC problem requires high precision, but usually deals with small sets of values, so the optimal domain for DTC analysis is different from that for problems which can do with less precision but cover huge ranges or sets of values. For maximal precision, one possibility is to convert abstract states into sets of concrete states, compute with transfer functions on these sets formed by applying the concrete transfer functions elementwise, and converting back to the original abstract domain when suitable. If the sets of concrete states do not grow too big, then this is tractable. Cofibered domains [15] provide a theoretical framework for this kind of use of multiple abstract domains.

References

1 The Atmel AVR. `http://en.wikipedia.org/wiki/Atmel_AVR`.

2 Gogul Balakrishnan and Thomas Reps. WYSINWYX: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, August 2010.

3 C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. In *Proc. Seventh International Workshop on Program Comprehension*, pages 192–199, 1999.

4 Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 3:165–199, 1989.

5 Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg, and Linus Källberg. ALF – a language for WCET flow analysis. In Niklas Holsti, editor, *WCET'09*, pages 1–11, Dublin, Ireland, June 2009. OCG.

6 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proc. 27th IEEE Real-Time Systems Symposium (RTSS'06)*, pages 57–66, Rio de Janeiro, Brazil, December 2006. IEEE Computer Society.

7 Niklas Holsti. Analysing switch-case tables by partial evaluation. In *Proc. 7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, 2007.

8 Niklas Holsti. Computing time as a program variable: a way around infeasible paths. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET'08)*, Prague, Czech Republic, July 2008.

9 Niklas Holsti, Jan Gustafsson, Linus Källberg, and Björn Lisper. Combining Bound-T and SWEET to analyse dynamic control flow in machine-code programs. Technical report, Mälardalen Real-Time Research Centre, Mälardalen University, November 2014.

10 Björn Lisper. SWEET – a tool for WCET flow analysis (extended abstract). In Tiziana Margaria and Bernhard Steffen, editors, *Proc. 6th International Symposium on Leveraging Applications of Formal Methods (ISOLA'14)*, volume 8803 of *Lecture Notes in Computer Science*, pages 482–485, Corfu, Crete, October 2014. Springer-Verlag.

11 Tidorum Ltd. Bound-T time and stack analyser. `http://www.bound-t.com`.

12 Hex-Rays SA. IDA disassembler. `https://www.hex-rays.com/products/ida`.

13 Rathijit Sen and Y. N. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *Proc. 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE'07)*, pages 39–48, Washington, DC, USA, 2007. IEEE Computer Society.

14 Henrik Theiling. Extracting safe and precise control flow from binaries. In *Proc. 7th International Conference on RealTime Computing Systems and Applications (RTCSA'00)*, pages 23–30, Cheju Island, South Korea, 2000. IEEE.

15 Arnaud Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In Radhia Cousot and David A. Schmidt, editors, *Proc. Third International Symposium on Static Analysis (SAS'96)*, volume 1145 of *Lecture Notes in Computer Science*, pages 366–382. Springer Berlin Heidelberg, Aachen, Germany, 1996.

16 W. Pugh et al. The Omega Project: Frameworks and algorithms for the analysis and transformation of scientific programs. `http://www.cs.umd.edu/projects/omega`.